FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Knowledge Specification for Automatic Performance Analysis APART Technical Report

Thomas Fahringer, Michael Gerndt, Graham Riley,
Jesper Larsson Träff

FZJ-ZAM-IB-9918

November 1999

(letzte Änderung: 05.11.99)

Knowledge Specification for Automatic Performance Analysis
APART Technical Report [1]

http://www.fz-juelich.de/apart

Workpackage 2
Identification and Formalization of Knowledge

Thomas Fahringer
Institute for Software Technology and Parallel Systems
University of Vienna
tf@par.univie.ac.at

Michael Gerndt
Central Institute for Applied Mathematics
Research Centre Juelich
m.gerndt@fz-juelich.de

Graham Riley
Department of Computer Science
University of Manchester
griley@cs.man.ac.uk

Jesper Larsson Träff
C&C Research Laboratories
NEC Europe Ltd.
traff@ccrl-nece.technopark.gmd.de

November 5, 1999

[1]The ESPRIT IV Working Group on *Automatic Performance Analysis: Resources and Tools* is funded under Contract No. 29488

Thanks

We would like to thank all our partners in the APART Working Group for their contribution to this topic. We would like to thank especially those members who participated in the discussions at the work package meeting at Malaga, September 1999: Maria Calzarossa (Università di Pavia), Antonio Espinosa (Universitat Autònoma de Barcelona), John Gurd (University of Manchester), Jarek Nabrzyskin (Pozman Supercomputing and Networking Center), Oscar Plata (Universidad de Málaga), and Emilio Zapata (Universidad de Málaga).

Abstract

The lack of a useful and accurate software infrastructure for measuring, modeling, and analyzing the performance of a wide variety of programming paradigms and architecture platforms is a critical issue for performance-oriented program development. Commonly, a cyclic process is employed to tune the performance of programs which includes the gathering of performance data through measurement and prediction and the analysis of the data collected on-the-fly or during a postmortem session to yield summary statistics and histories of program behavior. Usually, this process also involves comparison of the performance data with that of previous program versions. So far most approaches require the programmer to control this tedious, time-consuming, and error-prone process which is typically driven by some informal hypotheses about potential performance problems. Moreover, many tools are platform and language dependent and cannot correlate performance data gathered at lower levels (for example, from hardware counters) with higher-level programming paradigms. Further, they tend to focus only on specific program and machine behavior, and do not provide sufficient support to infer important performance properties.

In this report we describe a novel approach to the formalization of performance bottlenecks and the data required to detect them with the aim of supporting automatic performance analysis for a large variety of programming paradigms and architectures. We present the APART Specification Language (ASL) developed as part of the APART Esprit IV Working Group on *Automatic Performance Analysis: Resources and Tools*. This language allows the description of performance-related data through the provision of an object-oriented specification model and supports definition of performance properties in a novel formal notation. Performance-related data can either be static (gathered at compile-time, e.g. code regions, control and data flow information, predicted performance data, etc.) or dynamic (gathered at run-time, e.g. timing events, performance summaries, etc.) and is used as a basis for describing performance properties. A performance property (e.g. load imbalance, communication, cache misses, etc.) characterizes a specific type of performance behavior which may be present in a program. Checks for which properties are present in (the execution of) a program are given by a set of conditions defined over the performance-related data. Conditions have an associated confidence level which indicates the degree of certainty in the diagnosis of the presence of the performance property. Performance properties also have an associated severity measure (usually an expression), the magnitude of which specifies the importance of the property in terms of its contribution to limiting the performance of the program. The severity can be used to focus effort on the important performance issues during the (manual or automatic) performance tuning process.

Our approach is very general and can be efficiently employed to describe many performance properties for a large variety of programming paradigms and architectures. We illustrate our approach by applying it to some of the most important programming paradigms for performance-oriented scientific computing including MPI, OpenMP, and HPF.

Contents

Chapter 1

Introduction

1.1 Overview

Performance-oriented program development can be a daunting task. In order to achieve high or at least respectable performance on today's multiprocessor systems, careful attention to a plethora of system and programming paradigm details is required. Commonly, programmers go through many costly and time consuming cycles of experimentation involving the gathering and analysis (a-priori and post-mortem) of performance data, detection of performance problems, and code refinements. Clearly, the programmer must be intimately familiar with many aspects related to this experimental process. Although there exist a large number of tools to assist the programmer in performance experimentation, the responsibility for taking the majority of strategic decisions still lies with the programmer. It is particularly distressing that many performance tools remain platform and language dependent, cannot correlate performance data gathered at a lower level with higher-level programming paradigms, focus only on specific program and machine behavior, and do not provide sufficient support to infer important performance properties.

In this report we describe a novel approach to the task of formalizing the description of performance bottlenecks and the data required to detect them with the aim of supporting automatic performance analysis for a large variety of programming paradigms and architectures. This research has been performed as part of APART Esprit IV Working Group on *Automatic Performance Analysis: Resources and Tools* (APART, http://www.fz-juelich.de/apart).

In the remainder of this report we use the following terminology:

Performance-Related Data: Performance-related data defines information that can be used to describe performance properties of a program. There are two classes of performance related data. First, static data specifies information that can be determined without executing a program on a target machine. Static data is useful in order to specify dynamic performance related data and to formalize performance properties. Examples include code versions, program regions, source files, control and data flow information, loop scheduling information, predicted performance data, and information on the programming paradigm (e.g. master-slave, divide-and-conquer, bulk-synchronous, etc.). Second, dynamic performance related data describes the dynamic behavior of a program during execution on a target machine. This includes timing events, performance summaries and metrics, and communication patterns that are statically undetectable, etc.

Performance Property: A performance property (e.g. load imbalance, communication, cache misses, redundant computations, etc.) characterizes a specific performance behavior of a program and can be checked by a set of *conditions*. Conditions are associated with a *confidence value* (between 0 and 1) indicating the degree of confidence about the existence of a performance property. In addition, for every performance property a *severity measure* is provided the magnitude of which specifies the importance of the property. The severity can be used to focus effort on the important performance issues during the (manual or automatic) performance tuning process. Performance properties, confidence and severity are defined over performance-related data.

Performance Problem: A performance property is a performance problem, iff its severity is greater than a user- or tool-defined threshold.

Performance Bottleneck: A program has a unique performance bottleneck which is its most severe performance property. If this bottleneck is not a performance problem, then the program's performance is acceptable and does not need any further tuning.

For example, during performance analysis, a specific code region may be examined to determine the existence of a performance property denoted *communication*. The condition for this property holds if any process executing the region invokes communication (that is, if communication time for the region is greater than zero). The confidence value is 1, since measured communication time represents a proof for the presence of this property. The severity of the property may be calculated as the percentage of the communication time in the region relative to the execution time of the entire program. If the severity is above a user- or tool-defined threshold, then the communication performance property defines a performance problem. If this performance problem is the most severe of all the performance problems in the program, then it is the performance bottleneck. Commonly, a programmer may try to eliminate, or at least to alleviate, this bottleneck before examining any other performance problems.

This report introduces the APART Specification Language (ASL) which allows the description of performance-related data through the provision of an object-oriented specification model and which supports the definition of performance properties in a novel formal notation. Our object-oriented specification model is used to declare – without the need to compute – performance information. It is similar to Java, uses only single inheritance and does not require methods. A novel syntax has been introduced to specify performance properties.

The organization of this report is as follows. We continue this chapter with the presentation of an overall design of an automatic performance analysis environment that incorporates the specification of performance properties. Related work is discussed in Section 1.3. In the next chapter, Chapter 2, we describe our object oriented specification model for performance-related data by using UML (unified modeling language) class diagrams. We apply this model to some of the most important programming paradigms in current use including MPI, OpenMP, and HPF. In Chapter 3 we describe our syntax for performance property specification. Again, paradigm related property specifications are presented for MPI, OpenMP, and HPF. Conclusions and Future work are discussed in Chapter 4. In the Appendix we give an overview of UML, and summarize performance related data and performance properties.

1.2 Overall Design

Performance property specification as described in this report can be considered as part of a possible design for an automatic performance analysis environment. This environment comprises three components (see Figure 1.1):

Performance Property Specification defines information about performance properties for the current programming paradigm and machine, in combination with proof conditions and severity data.

Performance Process Specification reflects the knowledge applied in tuning the performance of programs including, for example, how many hypotheses about performance problems are evaluated. This evaluation can be based on stepwise refinement, i.e. the process specification determines which hypotheses are evaluated first before more precise hypotheses are examined. For example, it may be useful to prove that message passing is significant in a subroutine before examining individual MPI calls. More detailed analysis may require considerably more performance-related data.

Supplied Data Specification describes, for a particular tool, which of the performance-related data required for performance property specification can be obtained from that tool. Moreover, query commands to access this data can be indicated. Examples for such tools are PARADYN [MCCHI 95], SCALA [FaScPa 99], TAU [MoBrMa 94], and VAMPIR [NaArWeHoSo 96], etc. Based on the supplied data specifications, an automated performance analysis environment can use existing tools to access relevant data in the search for performance problems and bottlenecks.

An integrated system combining all three components should substantially alleviate the task of retargeting performance tools to new architectures and programming paradigms, facilitate the development of new performance tools and also enable the enhancement of existing tools by providing access to a wealth of performance information and analysis capabilities.

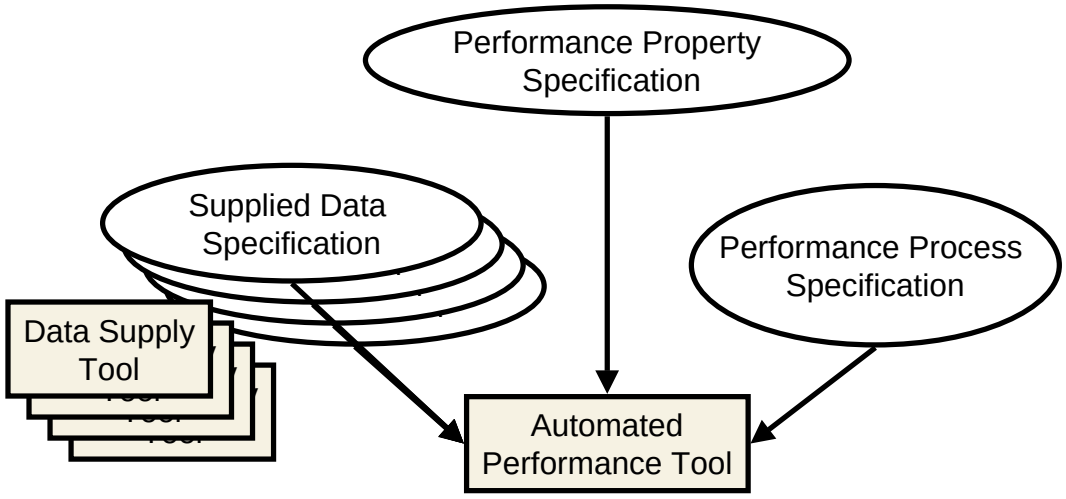

Figure 1.1: Design of an integrated automatic performance analysis environment.

1.3 Related Work

The specification of performance problems as presented in this report is a novel approach. Relatively few existing performance analysis tools and related projects apply specification languages in the context of performance analysis.

The most well-known automatic performance analysis tool is Paradyn [MCCHI 95]. Paradyn performs an automatic online analysis using dynamic instrumentation for monitoring. The *Performance Consultant* (PC) searches for performance bottlenecks according to the W^3 Search Model: each potential bottleneck is expressed in terms of *why* there is a problem, *where* in the application the problem is found (i.e., a focus), and *when* the problem occurs, i.e., in which phase(s) of the execution.

Hypotheses are conditions of the form *metric,focus* > *threshold*, where metric is a time-varying function that characterizes some aspect of a parallel program performance, such as CPU utilization or number of synchronization operations, and focus is the program location where the metric is measured. While metrics can be defined via the *Metric Description Language* (MDL) [Paradyn 98] the set of bottleneck hypotheses is currently predefined. It includes *CPUbound*, *Excessive Sync Waiting Time*, *ExcessiveIOBlockingTime*, and *TooManySmallIOOps*. The metric description specifies among other things the measurement basis (counter or timer), the aggregate operator (average, sum, minimum, or maximum) and the instrumentation actions.

A rule-based specification of performance bottlenecks and of the analysis process was developed within the context of the SVM-Fortran project [BeGeKr 96]. The performance analysis tool OPAL [GeKrOz 95] supports post-mortem analysis and used in combination with the monitoring system SAM, it realizes an incremental performance analysis process. New measurements can be requested based on the user's insight in the performance behavior. The measurements are executed during the next program run without having to recompile the code. Based on experience gained in applying this tool to real applications, a rule-based design for the automation of OPAL was developed [GeKr 97]. The rule base consists of a defined set of parameterized hypothesis with proof rules and refinement rules. The proof rules determine whether a hypothesis is valid based on the measured performance data. The refinement rules specify which new hypotheses are generated from a proven hypothesis. Therefore, the refinement rules specify the analysis process while the proof rules specify the knowledge about bottlenecks.

Another approach is to define a performance bottleneck as an event pattern or compound event which may occur during execution of a parallel program. Such patterns have to be detected in an event trace provided by tools like PAT [GaMo 98] after program termination. The compound event is built from primitive events such as those associated with entering a program region or sending a message.

EDL [BaWi 83] allows the definition of compound events based on extended regular expressions. Primitive events are clustered to higher-level events by certain formation operators. Relational expressions over

the attributes of the constituent events place additional constraints on valid event sequences obtained from the regular expression. Abstraction mechanisms allow the reuse of already defined event patterns to form custom hierarchies of events.

EARL [WoMo 98] describes event patterns in a more procedural fashion as scripts in a high-level event trace analysis language which may be implemented as an extension of common scripting languages like Tcl, Perl or Python. Frequently used, higher-level events like region instances or message transfers are represented by links between their constituent events, which can be easily traversed by a script. In addition, EARL supports navigation through function call stacks and message queues of a chosen execution state of the parallel program, enabling compact specification and efficient detection of the requested compound event.

Besides these tools and specification concepts for performance problems, a few others tools have been developed supporting automatic performance analysis.

KAPPA-PI [EsMaLu 98] is an automatic performance analyzer for PVM-programs developed at the Universitat Autonoma de Barcelona . It is a post-execution tool, implemented in PERL, that evaluates traces generated by the Tape/PVM monitoring library or by the VAMPIR MPI trace library. Based on a predefined list of performance bottlenecks, it searches for performance problems and their causes. In addition to trace data, it analyzes the source code using pattern matching.

One additional design, POIROT, was published by Robert Helm and Allen Malony [HeMa 95]. The design is based on the concept of heuristic classification. The main properties of a program run are extracted from trace data by a process called abstraction. These properties are matched against a database of possible performance bottlenecks and the selected bottleneck is refined to fit additional properties of the program run. Performance data are gathered via an environment interface that makes POIROT independent of intricate details of the programming environment, e.g., how to instrument a program.

FINESSE [Mu 99, Mu 00] is a prototype environment designed to support rapid development of parallel programs for single-address-space computers by both expert and non-expert programmers. The environment provides semi-automatic support for systematic, feedback-guided identification and reduction of the various classes of overhead associated with parallel execution. FINESSE automatically identifies code regions with significant overhead, classifies and quantifies this overhead and ranks regions according to their execution time. FINESSE also suggests possible improvements which should lead to improved implementations.

P^3T [Fa 95, Fa 96] is a static performance estimator for data parallel programs which guides the selection of efficient data distribution strategies and profitable code transformations. This tool tries to answer three fundamental questions: 1. What performance bottlenecks exist? 2. Where are these these performance bottlenecks in the input program? 3. What must be done in order to gain performance? P^3T automatically computes a variety of performance parameters including work distribution, number of data transfers, amount of data transferred, transfer times, network contention, number of cache misses, and computation times. Through a graphical user interface the programmer can optionally specify for each of these parameters a specific threshold which implies a bottleneck. The default option is that P^3T visualizes every computed performance parameter relative to the worst-case value found in the entire program. Color-coded performance visualization directly guides the user to all bottlenecks of a program found by P^3T. A list of code transformations is suggested to eliminate or alleviate each specific performance bottleneck.

Chapter 2

Performance Related Data Specification

A necessary prerequisite for automatic performance analysis is the availability of sufficient information to allow the examination of the performance behavior of an application on a given architecture.

Performance-related data can be obtained either statically, by using, for instance, program analysis tools, or dynamically with the aid of monitoring tools. Dynamic information collection requires the application to be executed on a target machine, whereas accessing static information does not. Examples of static data include, code versions, program regions, source files, control and data flow information, predicted performance data, and information on the programming paradigm (e.g. master-slave, divide-and-conquer, bulk-synchronous, etc.). Timing events, performance summaries and metrics, and communication patterns that are statically undetectable represent dynamic information.

A key issue in our work on automatic performance analysis is to find a well-suited representation of static and dynamic information such that we can exploit performance-related data obtained from many different sources (e.g. performance tools, program analysis tools, databases, user provided data, etc.). Moreover, it is of paramount importance to be able to relate performance information derived from different sources. These abilities will alleviate current difficulties encountered in the specification of performance properties and support the task of automatically searching for performance bottlenecks.

In this section, we introduce a specification language for describing performance-related data. We first present class libraries, in UML notation (see Appendix A), for performance-related data that is programming paradigm independent. Thereafter, specialized class libraries and examples are presented for the following programming paradigms: MPI, HPF, and OpenMP.

2.1 Specification Language

Figure 2.1 shows the syntax for specifying both static and dynamic performance-related data in Backus Naur form. Performance-related data is described by a set of classes following an object-oriented style with single-inheritance. Among others, class members can be of type FLOAT (e.g., for timing measurements), BOOLEAN (e.g., for flags), INT (e.g., for counting events), STRING (e.g., for naming applications or files), DATETIME (time at which some event occurs), and *reference* (e.g. for named enum types and class names). An identifier is described by *ident*. *SETOF* and *ENUM* enable set and enumeration notations.

Syntax variables in the syntax diagrams ending with "-list" identify a colon-separated list of one or more elements. For example, *string-list* represents a list of character constants such as "DO, FORALL, WHILE".

2.2 Standard Class Library

In this section we describe a library of classes that represent static and dynamic information for performance property analysis. We distinguish two sets of classes. First, the set of base classes which

class-def	**is**	`CLASS` *ident* [`EXTENDS` *ident*]'{' *member-def* $*$ '}' ';'
member-def	**is**	*type ident* ';'
type	**is**	`FLOAT`
	or	`BOOLEAN`
	or	`INT`
	or	`STRING`
	or	`DATETIME`
	or	*set-type*
	or	*enum-type*
	or	*reference*
set-type	**is**	`SETOF` *type*
enum-type	**is**	`ENUM` *ident* '{' *string-list* '}'

Figure 2.1: Syntax for describing performance-related data.

is independent of any programming paradigm, and secondly, programming paradigm-dependent classes comprising shared memory, data parallel, and message passing paradigms. The programming paradigm dependent classes are shown for HPF (High Performance Fortran [HPF 93]), OpenMP (shared memory paradigm [DaMe 98]), and MPI (Message Passing Interface [SnOtHUWaDo 98]) which are implementations of the shared memory, data parallel and message passing paradigm, respectively.

Note that we expect that most data models described with this language will have a similar overall structure. This similarity was captured in the design of the base classes. Future data models can build specialized classes in form of subclasses.

The data models presented for MPI, HPF, and OpenMP are intended to be examples. They cover typical program, machine, and performance data available in programming environments for those paradigms. Since they have been defined without a concrete programming environment with performance tools providing performance-related data in mind, it is possible, that those models do not fit the readers favorite programming environment.

Note that we do not claim that our class library is complete. Our classes, and in particular their attributes, can be extended to include other static and dynamic information to model relevant performance aspects of a large variety of programming paradigms.

Figure 2.2 shows the UML representation of the base classes which are programming paradigm independent. Initially, there is an application for which performance analysis has to be done. Every application has a name and may possibly have a number of implementations, each with a unique version number. Versions may differ with respect to their source files and experiments. Every source file (the contents of which are stored in a generic string) has one or several static code regions each of which is uniquely specified by *start_pos* (position where region begins in the source file) and *end_pos* (position where region ends in the source file). A position in a region is defined by a line and column number with respect to the given source file. Among other things, regions can be continuous sequences (e.g., entire programs or loops) or parts (e.g., expressions or even data references) of source-code lines. Regions can have sub-regions. For instance, we can represent all procedures or loops of a region by using its *sub_regions* attribute.

Experiments – denoting the second attribute of a version – are described by the time (*start_time*) when the experiment started and the number of processors (*nr_processors*) that were available to execute the version. Furthermore, an experiment is also associated with a static description of the machine (e.g. number of processors available) that is used for the experiment. Every experiment includes also dynamic data, i.e. a set of region summaries (*profile*) and a set of events (*trace*). The class *RegionSummary* describes performance information across all processes employed for the experiment. Region summaries are associated with the appropriate region. The class *Events* represents information about individual events occurring at runtime, such as sending a message to another process. Each event has a *time_stamp* attribute determining when the event occurred and a *process* attribute determining in which process the event occurred.

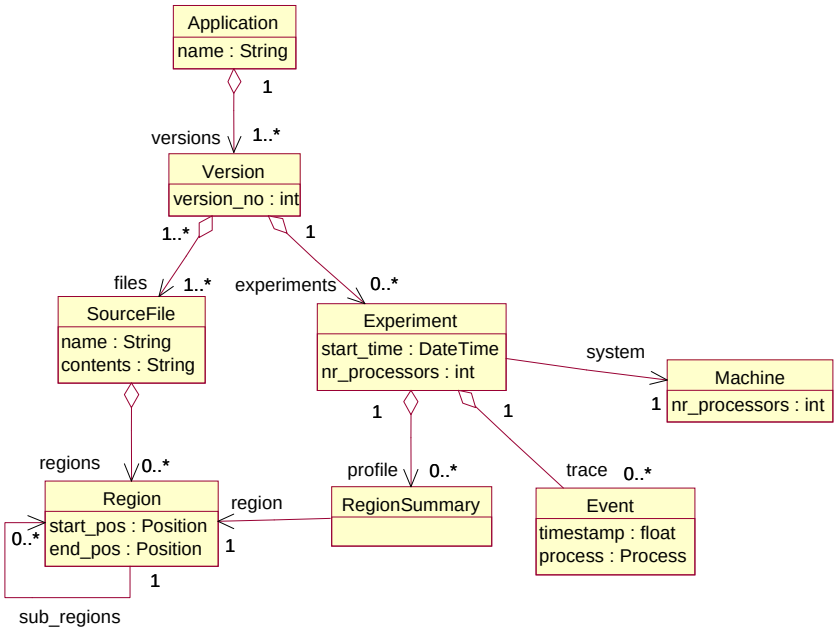

Figure 2.2: Base classes of performance-related data models.

2.3 Paradigm Specific Data Models

2.3.1 MPI Classes

In this section we describe static and dynamic information for MPI which is an implementation of the message passing paradigm. Figure 2.3 outlines the classes for static MPI regions. Class *MPIRegion* is a subclass of Region (see Figure 2.2) and contains two attributes: *paradigm* and *role* which, respectively, relate to the paradigm implemented (e.g. master-slave, divide-and-conquer, and bulk-synchronous) and to the role (e.g. master/slave send/receive operation) of a given region in a paradigm. *MPIRegion* is further refined to:

- *LoopRegion* specifies different loop types such as DO, WHILE, or FORALL loop.
- *CollPrimitive* refers to various collective operations. This class comprises an attribute *type* for the type of collective operation (e.g. reduction or broadcast), and an attribute *sync* for a specific synchronization mode (e.g. barrier or nobarrier).
- *PointToPointPrimitive* provides more specific information about the point-to-point communication. An attribute *type* determines whether the underlying communication is based on a send or receive operation. The communication mode (e.g. buffered, synchronous, ready) is denoted by attribute *mode*. Blocking or nonblocking communication can be defined by attribute *semantics*.
- *MPIio* provides information about MPI Input/Output routines.

The precise semantics of the above mentioned MPI communication modes and types can be found in [SnOtHUWaDo 98].

Figures 2.4 and 2.5, respectively, describe summary and event information which reflects the dynamic behavior of MPI. Class *MPIRegionSummary* in Figure 2.4 extends *RegionSummary* (see Figure 2.2) and reflects dynamic performance information across all processes that execute a region. *MPIRegionSummary* has two attributes: *sums* and *process_sums* which, respectively, describe summary information for a specific region across all processes and for an individual process. The attributes of class *MPISummary* are given as sums across all processes with respect to all instances of a specific region:

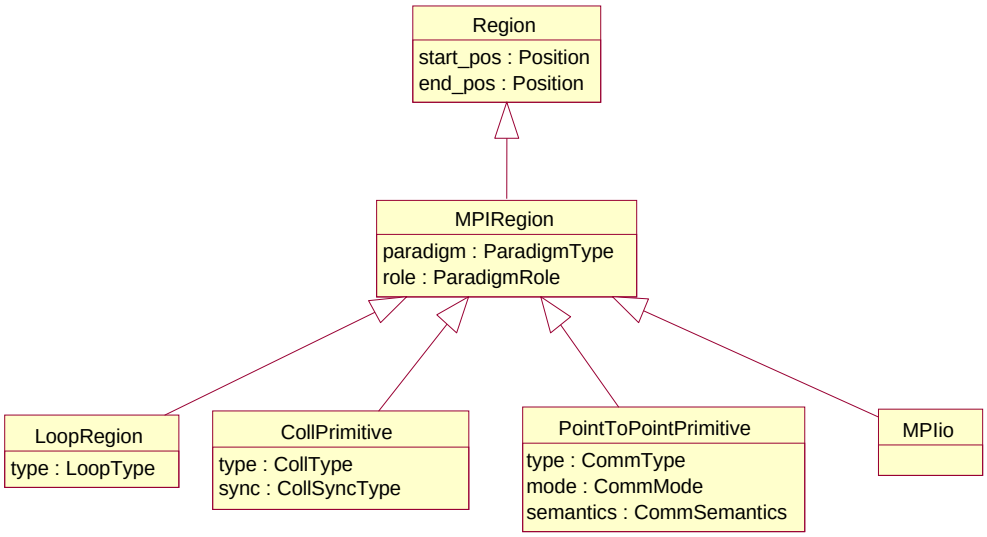

Figure 2.3: Regions in the MPI data model.

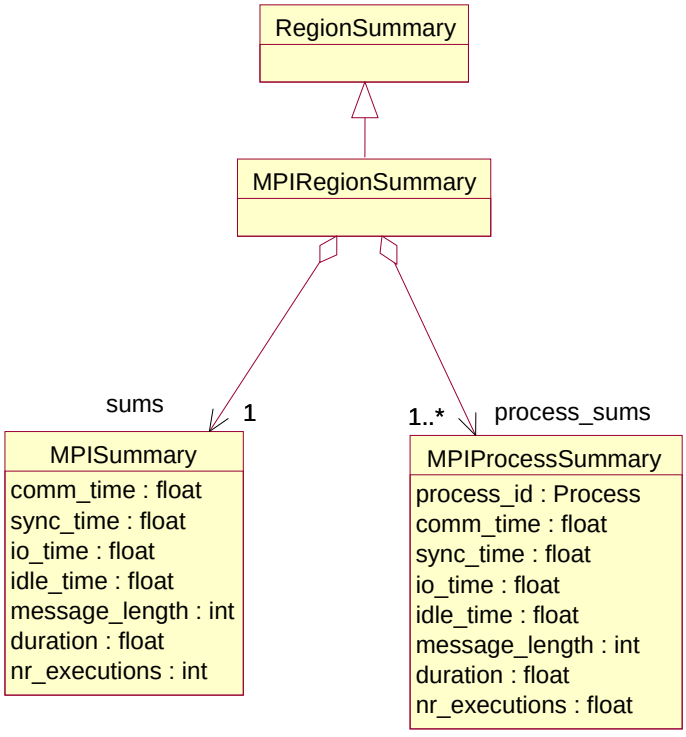

Figure 2.4: Summaries in the MPI data model

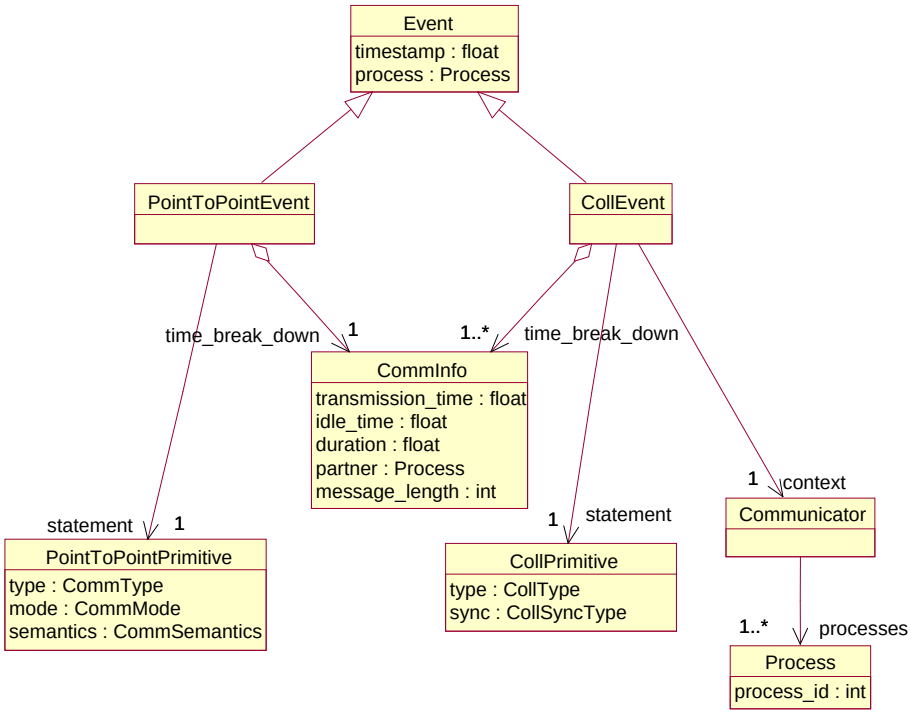

Figure 2.5: Events in the MPI data model.

- *comm_time*: communication time
- *sync_time*: barrier synchronization time
- *io_time*: input/output time
- *idle_time*: idle time
- *message_length*: sum of the length of all messages sent
- *duration*: execution time
- *nr_executions*: number of executions of a given region

Class *MPIProcessSummary* relates to region summary information for a specific process identified by attribute process.

Figure 2.5 describes dynamic events for MPI programs. Two specialized subclasses of the event base class are introduced for MPI: *PointToPointEvent* and *CollEvent* which, respectively, relate to point-to-point communication and collective operation events. Both classes inherit the timestamp and the process attribute. The *PointToPointEvent* has two additional attributes that specify the primitive (class *PointToPointPrimitive*) which causes the event and provide detailed timings information (class *CommInfo*) for the event. Class *CommInfo* comprises the following attributes:

- *transmission_time*: time for transferring data
- *idle_time*: waiting time
- *duration*: total execution time which is the sum of *idle_time* and *transmission_time*
- *partner*: target process of communication
- *message_length*: length of message

Class *CollEvent* relates to collective operations which is further described by the collective primitive (see Figure 2.3), the MPI context (set of processes that jointly execute the collective operation), and detailed timings information. The event includes for each communication partner in the collective operation a separate *CommInfo* object.

2.3.2 HPF Classes

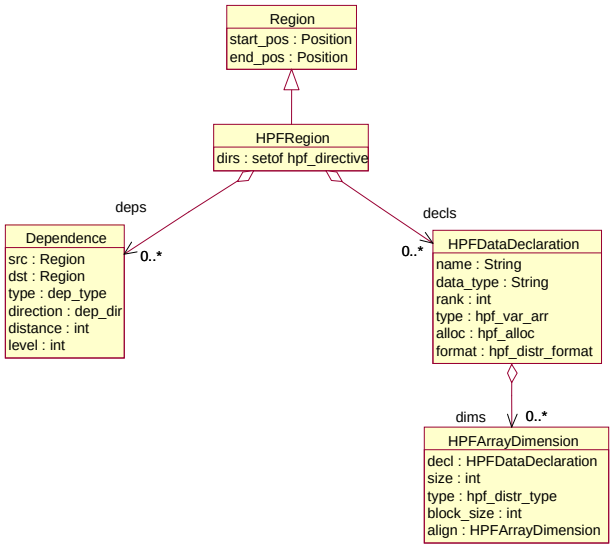

Figure 2.6: Static performance-related information for *HPFRegions*.

In this section we describe the class libraries for HPF which is an implementation of the data parallel programming paradigm. Class *HPFRegion* extends Region (see Figure 2.2) and comprises the following attributes (Figure 2.7) representing static performance-related information:

- *dirs* describes HPF directives such as PROCESSORS, DISTRIBUTED, ALIGN, RESHAPE, INDEPENDENT, etc.
- *deps* specifies data dependences implied by code regions.
- *decls* specifies HPF data declarations for scalars and arrays. Attribute *alloc* denotes whether data has been declared DYNAMIC or STATIC. For arrays there is additional data provided for every dimension including size of dimension, distribution and alignment information.

Figure 2.7 displays several subclasses which extend *HPFRegion*:

- *HPFProcedure*:
- *HPFLoop*:
- *HPFIfBlock*:
- *HPFBasicBlock*:
- *HPFProcedureCall*:
- *HPFArrayAssignment*:

Class *HPFLoop* can be further specified by attribute ltype (e.g. DO, INDEPENDENT, and FORALL). Dynamic performance-related data is described by class *HPFRegionSummary* in Figure 2.8 with the following attributes:

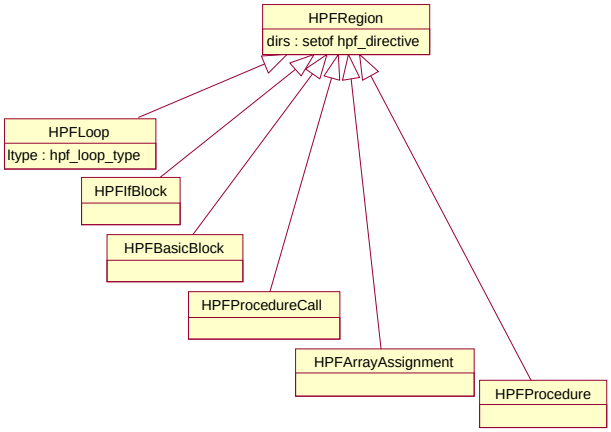

Figure 2.7: Subclasses of *HPFRegion*.

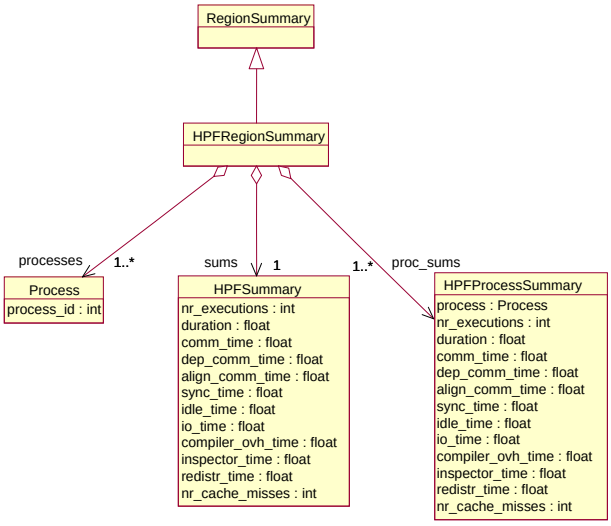

Figure 2.8: Dynamic performance-related information (summaries) in the HPF data model.

- *processes* specifies the set of processes executing a region
- *sums* reflects performance summary information across all processes executing the region
- *proc_sums* indicates performance summary information for a region with respect to individual processes.

Class *HPFSummary* contains several performance attributes which are average values across all processes with respect to a specific region:

- *nr_executions*: number of times the region has been executed
- *duration*: time spent in executing the region
- *comm_time*: communication time
- *dep_comm_time*: communication time caused by data dependences
- *align_comm_time*: communication time caused by data alignment
- *comm_time*: communication time
- *sync_time*: synchronization time
- *idle_time*: idle time
- *io_time*: input/output time
- *compiler_ovh_time*: compiler overhead time
- *inspector_time*: time spent in inspector/executor phase (compiler inserted code to handle irregular problems)
- *redistr_time*: time spent in redistribution of arrays
- *nr_cache_misses*: number of cache misses.

Class *HPFProcessSummary* contains all attributes of class *HPFSummary* restricted to average values for a specific process.

2.3.3 OpenMP Classes

Figure 2.9 shows the classes that model static information for OpenMP programs. Class *SmRegion* is a subclass of *Region* (see Figure 2.2) and contains an attribute with data dependence information about the modeled region. *SmRegion* is then further refined by two subclasses *ParallelRegion* and *SequentialRegion* which, respectively, describe parallel and sequential regions. Note that in OpenMP a master thread is responsible for the execution of sequential regions and is also responsible for engaging other threads in the execution of parallel regions. Typically, for efficiency reasons, threads will sit in some form of idle pool while the master executes sequential regions, rather than threads being continually created and destroyed by the master in an explicit fork/join model. Currently we have defined four sequential regions including:

- *Function*
- *IfBlock*
- *BasicBlock*, and
- *FunctionCall*

which respectively, refer to a function, IF-THEN-ELSE construct, basic block (single-entry-exit code regions – [AhSeUl 88]), and a function call in the OpenMP program. When a master thread encounters a parallel region it releases a set of threads from an idle pool (they typically are waiting on a region entry barrier) in order to execute the region in parallel. Parallel regions include a boolean variable *no_wait_exit* which denotes whether or not the region is terminated by an explicit exit barrier operation. A specific execution of a region corresponds to a region instance. The following parallel regions are modeled:

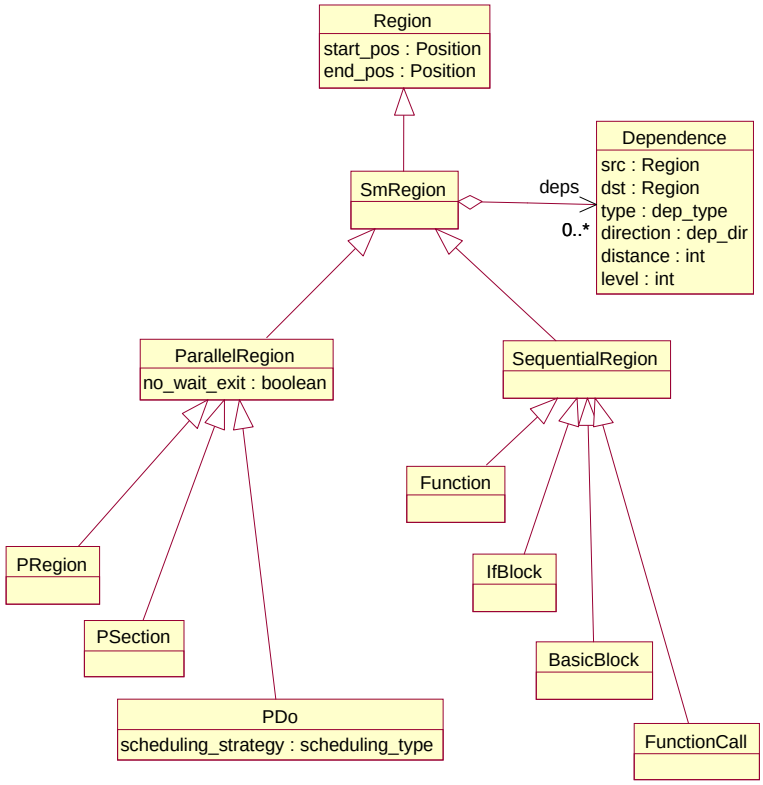

Figure 2.9: OpenMP classes for static information

- *PRegion* correspond to OpenMP's parallel region which is a block of code whose instances are executed by all threads in a replicated mode.
- *PSection* refer to OpenMP's parallel sections each of which is executed by a specific thread in parallel.
- *PDo* relates to OpenMP's parallel DO construct whose iterations are executed by a set of threads in parallel. The DO loop's iterations can be distributed in various ways including STATIC(CHUNK), DYNAMIC(CHUNK), and GUIDED(CHUNK) onto the set of threads (as defined in the OpenMP standard). The distribution is specified in class *PDo*. STATIC(CHUNK) distribution means that the set of iterations are consecutively distributed onto the threads in blocks of CHUNK size (resulting in block and cyclic distributions). DYNAMIC(CHUNK) distribution implies that iterations are distributed in blocks of CHUNK size to threads on a first-come-first-served basis. GUIDED (CHUNK) means that blocks of exponentially decreasing size are assigned on a first-come-first-served basis. The size of the smallest block is determined by CHUNK size.

Figure 2.10 shows the OpenMP class library for dynamic information. Class *SmRegionSummary* extends class *RegionSummary* (see Figure 2.2) and comprises three attributes: *nr_executions* specifies the number of times a region has been executed by the master thread, *sums* describes summary information across all region instances, and *instance_sums* relates to summary information for a specific region instance. The attributes of class *SmSums* include:

- *duration*: time needed to execute region by master thread
- *non_parallelized_code*: time needed to execute non-parallelized code
- *seq_fraction*: $\frac{non\_parallelized\_code}{duration}$
- *nr_remote_accesses*: number of accesses to remote memory by load and store operations in ccNUMA machines [CuSiGu 99]

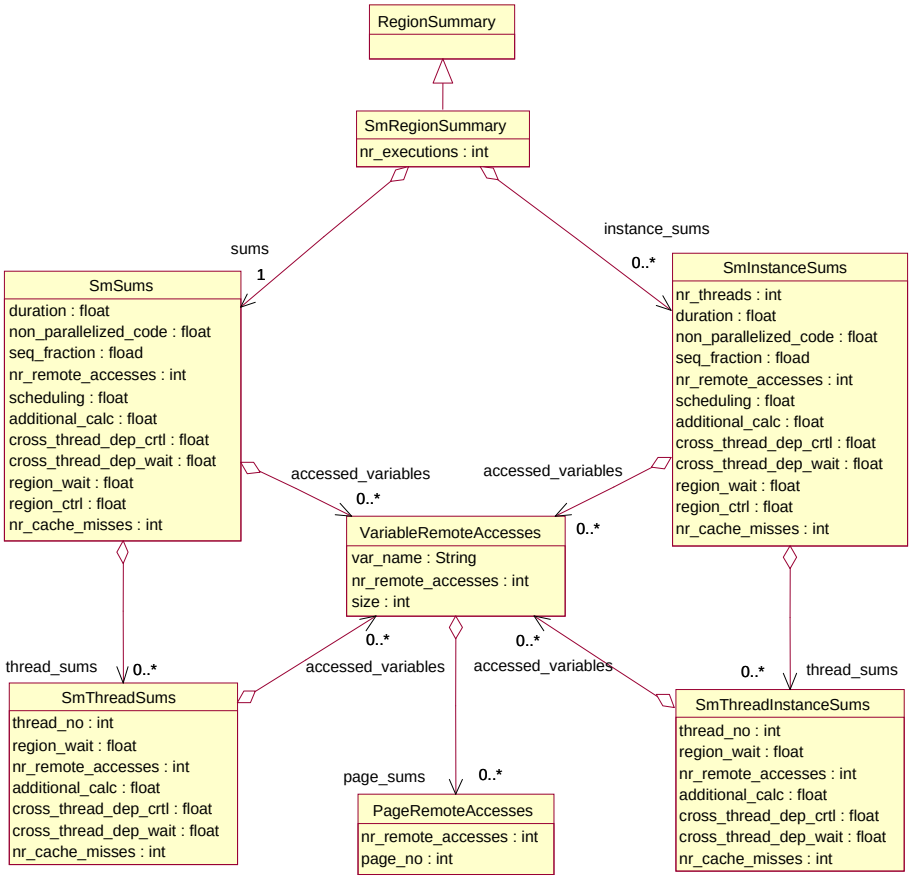

Figure 2.10: OpenMP classes for dynamic information

- *scheduling*: time needed for scheduling operations (e.g. scheduling of threads)
- *additional_calc*: time needed for additional computations in parallelized code (e.g. to enforce a specific distribution of loop iterations) or for additional computations (e.g. where it is cheaper for all threads to compute a value rather than communicate it, possibly with synchronization costs)
- *cross_thread_dep_ctrl*: synchronization time except for entry and exit barriers and waiting in locks
- *cross_thread_dep_wait*: synchronization waiting time except waiting in entry or exit barrier
- *region_wait*: waiting time in entry or exit barrier
- *region_ctrl*: time needed to execute region control instructions (e.g. controlling barriers)
- *nr_cache_misses*: number of cache misses
- *thread_sums*: summary data for every thread executing the region
- *accessed_variables*: set of remote access counts for individual variables referenced in that region

Note that attributes *duration* and *region_ctrl* are given with respect to the master thread, whereas all other attributes are average values across all threads that execute a region. Summary data (described by class *SmThreadSums*) for every thread executing a region is specified by *thread_sums* in *SmSums*. The attributes of *SmThreadSums* are a subset of class *SmSums* attributes and refer to summary information for specific threads identified by a unique thread number (*thread_no*).

In addition to the number of remote accesses in a region, the number of remote accesses is collected for individual variables that are referenced in that region. This information is modeled by the class *VariableRemoteAccesses* with the attributes *var_name*, *nr_remote_accesses*, and *size* which denotes the total size of the variable in bytes. This information can be measured if address range specific monitoring is supported, e.g. [KaLeObWa 98]. The last attribute of this class is *page_sums* which is a set of page-level remote access counters. For example, the remote access counters on SGI Origin 2000 provide such information [CuSiGu 99]. With the help of additional mapping information, i.e. mapping variables to addresses, this information can be related back to program variables. Each object of class *PageRemoteAccesses* determines the *page_no* and the number of remote accesses.

The second attribute of *SmRegionSummary* is given by *instance_sums* which is described by a class *SmInstanceSums*. This class specifies summary information for a specific region instance. *SmInstanceSums* contains all attributes of *SmSums* and the number of threads executing the region instance. Finally, class *SmThreadInstanceSums* describes summary information for a given region instance with respect to individual threads.

Chapter 3

Performance Property Specification

A performance property characterizes an aspect of the dynamic behavior of an application. In the context of automatic performance analysis we need to specify only performance properties describing inefficient behavior.

A performance property typically occurs in a specific context. This context can include the program region, a specific process, or a specific instance of that region in a specific process. For example, a property *message passing* can exist for specific regions, such as a subroutine and a message passing statement, or for a specific instance of a region, e.g. an instance of a subroutine in a process.

The existence of a property can be checked by evaluating appropriate conditions based on static and dynamic performance-related data. It is clearly possible that different conditions might exist at the same time, especially if conditions may only give an indication for the existence of that property. Whether a condition proves the existence or indicates the existence is determined by the confidence expression of the property (see below).

Which condition is evaluated might depend on the required information. For example, on shared virtual memory machines thrashing of pages is a major problem. A very good indication for thrashing is a big number of page faults. But, this is not a proof. To be able to prove its existence, individual page fault events have to be traced which may perturbate program execution much more than profiling and huge amounts of data can be generated.

As mentioned above, for each condition, a confidence value between 0 and 1 indicates the confidence in this check. A tool might use this information to first do a fast and simple check with a lower confidence based on already existing information before requesting more detailed information.

The last feature of a performance property is the severity expression. It returns a value indicating the importance of the property in relation to other performance properties.

Performance properties of parallel programs belong to different categories. For example, synchronization and message passing belong to the execution time category, while memory overhead belongs to the memory category. The severity expression of the properties in a single category can easily be normalized so that a global ranking of those properties is possible. For properties of different categories it is difficult or impossible to do that. We currently favor the concept of having global conversion functions between categories. Those conversion functions could easily be adapted to the programmer's preferences while the individual severity expressions need not be changed.

3.1 Specification Language

This section introduces the syntax constructs of the APART specification language for specifying performance properties. The syntax for the structure of the whole specification is shown in Figure 3.1. The specification consists of the performance-related data model specification followed by the performance property specification.

The performance property part consists of a set of global definitions followed by the property specifications. The definitions specify functions or constants that can be used in the individual property specifications to make them more readable.

performance-property-spec **is** PERFORMANCE DATA
class-def ∗
PERFORMANCE PROPERTIES
[LET
def ∗
IN]
property ∗
END

Figure 3.1: Overall structure of the specification including the performance-related data model and the performance properties.

property **is** PROPERTY *pp-name* '(' *arg-list* ')' '{'
[LET
def ∗
IN]
pp-condition
pp-confidence
pp-severity
'};'
arg **is** *type ident*

pp-condition **is** CONDITION ':' *conditions* ';'
conditions **is** *condition*
or *condition* OR *conditions*
condition **is** ['(' *cond-id* ')']*bool-expr*

pp-confidence **is** CONFIDENCE ':' MAX '(' *confidence-list* ')' ';'
or CONFIDENCE ':' *confidence* ';'
confidence **is** ['(' *cond-id* ')' '->'] *arith-expr*

pp-severity **is** SEVERITY ':' MAX '(' *severity-list* ')' ';'
or SEVERITY ':' *severity* ';'
severity **is** ['(' *cond-id* ')' '->'] *arith-expr*

Figure 3.2: Property specification syntax.

def	**is**	*function-def*
	or	*const-def*
function-def	**is**	*type ident* '(' *arg-list* ')' '=' *expr* ';'
const-def	**is**	*type ident* '=' *expr* ';'
expr	**is**	*set-expr*
	or	*arith-expr*
	or	*bool-expr*

Figure 3.3: Definition of functions and constants facilitating subsequent specifications.

The property specification (Figure 3.2) defines the name of the property, its context via a list of parameters, and the condition, confidence, and severity expressions. Each property specification can also include local definitions that are then available in subsequent specifications of the property.

The property specification is based on a set of parameters. These parameters specify the property's context and parameterize the expressions.

While the context of a property was defined above, the use of additional parameters will be shown by an example. The severity specification will typically be based on a parameter specifying the ranking basis (*rank_basis*). If, for example, a representative test run of the application has been monitored, the time spent in message passing should be compared to the total execution time. If, instead, a short test run is the basis for performance evaluation since the application has a cyclic behavior, the message passing overhead should be compared to the execution time of the shortened loop.

The condition specification consists of a list of conditions. A condition is a predicate that can be prefixed by a condition identifier (*cond-id*). The identifiers have to be unique with respect to the property since the confidence and severity specifications can refer to the conditions via those condition identifiers.

The confidence specification is an expression that computes the maximum of a list of confidence values. Each confidence value is computed via an arithmetic expression resulting in a value in the interval of 0 and 1. The value can be guarded by a condition identifier introduced in the condition specification. The condition identifier represents the value of the condition. This confidence value is computed only if the condition evaluates to TRUE.

The severity specification has the same structure as the confidence specification. It computes the maximum of the individual severity values of the conditions.

Figure 3.3 specifies the syntax of definitions. Definitions can be local to a property or global to all properties. Two types of definitions are allowed: definitions of functions and constants. The right-hand side of both definitions are boolean, arithmetic, or set expressions.

Figure 3.4 defines the syntactical structure of boolean expression, also called predicates. Predicates can be built from other predicates with the standard boolean operations. An atomic predicate is either a reference to a boolean attribute or an external function returning a boolean value. Functions can be supplied by the environment. For example, testing predicates based on a trace might require pattern matching. Such predicates can be implemented via external functions and thus, pattern matching for traces need not be integrated into the language. Information in the data model can be accessed via references.

Set expressions (Figure 3.5) can also be built with standard operations. Here, the terminal symbols '+', '/', and '-' denote set union, set intersection and subtraction, respectively. The language also supports the union and intersection operators for sets of sets, as well as the UNIQUE function. It selects a unique value from the set given as argument.

Figure 3.6 introduces the syntax of arithmetic expressions. In addition to the standard syntax, we allow arithmetic operations that work on elements of sets.

3.2 Paradigm Related Property Specification

This section presents the current set of performance properties for the selected programming paradigms. This set is not intended to be a full catalog of performance properties but is a collection of typical examples showing the applicability of the language features.

bool-expr	**is**	*bool-expr* `AND` *bool-expr*
	or	*bool-expr* `OR` *bool-expr*
	or	`NOT` *bool-expr*
	or	'(' *bool-expr* ')'
	or	*quantifier-list* `SUCH THAT` *bool-expr*
	or	*function-name* '(' *expr-list* ')'
	or	*reference*
	or	*arith-expr relop arith-expr*
quantifier	**is**	`FORALL` *bound-variable-list*
	or	`EXISTS` *bound-variable-list*
	or	`NEXISTS` *bound-variable-list*
bound-variables	**is**	*ident-list* `IN` *set-expr*
relop	**is**	'>'
	or	'<'
	or	'=='
	or	'! ='
	or	'>='
	or	'<='
reference	**is**	*ident*
	or	*ident*'.'*reference*

Figure 3.4: Syntax for predicates.

set-expr	**is**	*set-expr* '+' *set-expr*
	or	*set-expr* '/' *set-expr*
	or	*set-expr* '-' *set-expr*
	or	'(' *set-expr* ')'
	or	*reference*
	or	*function-name* '(' *parm-list* ')'
	or	'{' [*set-expr* `WHERE`]*ident* `IN` *set-expr* [`WITH` *bool-expr*]'}'
	or	*set-op* '(' *set-expr* ')'
set-op	**is**	`UNION`
	or	`INTERSECTION`
	or	`UNIQUE`

Figure 3.5: Syntax for set expressions.

arith-expr	**is**	*arith-expr* '+' *arith-expr*
	or	*arith-expr* '-' *arith-expr*
	or	*arith-expr* '*' *arith-expr*
	or	*arith-expr* '/' *arith-expr*
	or	'(' *arith-expr* ')'
	or	*reference*
	or	*function-name* '(' *parm-list* ')'
	or	*arith-set-op* '(' *arith-expr* `WHERE` *ident* `IN` *set-expr* ')'
arith-set-op	**is**	`SUM`
	or	`COUNT`
	or	`MAX`
	or	`MIN`
	or	`STDEV`

Figure 3.6: Syntax for arithmetic expressions.

3.2.1 MPI

This section demonstrates the features of the APART specification language in the context of the message passing paradigm. Although most of the properties are independent of the specific message passing library, the terminology used is based on MPI.

The following performance properties are presented:

- costs
- communication_costs
- synchronization_costs
- io_costs
- dominating_communication
- frequent_communication
- big_messages
- late_sender
- late_receiver
- uneven_mp_distribution
- load_imbalance_at_barrier
- slow_slaves
- overloaded_master

3.2.1.1 MPI global definitions

```
MPIRegionSummary summary(MPIRegion r, Experiment e)=
                     UNIQUE({sumr IN e.profile WITH sumr.region==r});

float duration(MPIRegion r, Experiment e)=summary(r,e).sums.duration;
```

In most property specifications it is necessary to access the summary data of a given region for a given experiment. Therefore, we defined the summary function that returns the appropriate *MPIRegionSummary* object. It is based on the set operation *UNIQUE* that selects arbitrarily one element from the set argument which has cardinality one due to the design of the data model. By the design of the data model, the set in the expression above is indeed singleton.

The second function determines the execution time of the region in the given experiment. The return value is the sum of the individual execution times of all MPI processes.

For all MPI performance properties the severity is computed by relating some aspect of the the execution time to the duration of a given *rank_basis* region in the experiment.

3.2.1.2 costs

```
property costs(MPIRegion r, Experiment e, Region rank_basis){
LET
  float CostSum = summary(r,e).sums.comm_time +
                  summary(r,e).sums.sync_time +
                  summary(r,e).sums.io_time;
IN
  CONDITION:   CostSum>0;
  CONFIDENCE:  1;
  SEVERITY:    CostSum/duration(rank_basis,e);
}
```

The most general performance property characterizes the region as having some performance overheads or costs. The costs of a region can be subdivided into time for communication, time for synchronization, i.e. barrier synchronization, and time for I/O. The region has this property if CostSum is greater than 0. Clearly the confidence in that condition is one.

The severity of this property is the fraction of the time spent for costs compared to the duration of ranking basis, typically the duration of the main program. Note, that *comm_time*, *sync_time*, *io_time*, and *duration* are sums of the time spent in each process.

The severity of this property may be larger than the severity of the individual properties for each of the categories. This may lead to the selection of the cost property as a performance problem according to the predefined severity threshold while the individual properties, i.e. *communication_costs*, *synchronization_costs*, and *io_costs* may not be marked as performance problems.

3.2.1.3 communication_costs

```
property communication_costs (MPIRegion r, Experiment e, Region rank_basis){
LET
  float cost = summary(r,e).sums.comm_time;
IN
  CONDITION:   cost>0;
  CONFIDENCE:  1;
  SEVERITY:    cost/duration(rank_basis,e);
}
```

This property determines whether a region includes communication. Its condition and severity is based on the appropriate global sums in the performance-related data model. The severity is the fraction of the communication costs in relation to the execution time of *rank_basis*.

3.2.1.4 synchronization_costs

```
property synchronization_costs (Region r, Experiment e, Region rank_basis){

LET
  float barrier_time = summary(r,e).sums.sync_time;
IN
  CONDITION:   barrier_time > 0;
  CONFIDENCE: 1;
  SEVERITY:   barrier_time/duration(rank_basis,e)
}
```

Synchronization_costs is a property of a region if any process spends some time in barrier synchronization. The severity is the fraction of the synchronization costs in relation to the execution time of *rank_basis*.

3.2.1.5 io_costs

```
property io_costs (Region r, Experiment e, Region rank_basis){

LET
  float io_time = summary(r,e).sums.io_time;
IN
  CONDITION:   io_time > 0;
  CONFIDENCE: 1;
  SEVERITY:   io_time/duration(rank_basis,e)
}
```

Io_costs is a property of a region if any process spends some time in input/output. The severity is the fraction of the io costs in relation to the *rank_basis*.

3.2.1.6 dominating_communication

```
property dominating_communication(MPIRegion r, Experiment e, Region rank_basis){
LET
  setof MPIRegionSummary comm_summaries=
           {x IN e.profile
            WITH
              typeof(x.region)==PointToPointPrimitive
            OR
              (typeof(x.region)==CollPrimitive AND x.region.type != Barrier)
           };
  float max_comm_time = MAX(sum.sums.duration WHERE sum IN CommSummaries);
IN
  CONDITION: (typeof(r)==PointToPointPrimitive OR
              (typeof(r)==CollPrimitive AND x.region.type != Barrier)) AND
              summary(r,e).sums.duration==max_comm_time;
  CONFIDENCE: 1;
  SEVERITY: max_comm_time/duration(rank_basis,e);
}
```

The call site of an MPI routine with the maximum communication time has the *dominating_communication* property. The constant *comm_summaries* specifies the set of summary objects of communication statements, i.e. it excludes call sites of *MPI_barrier*. The constant *max_comm_time* is the maximum of the execution time of all call sites in *comm_summaries*.

typeof(obj) denotes the type of a class such that obj is an instance of that class.

The condition of this property checks whether the region is a communication statement but not a barrier call, and whether its duration is equal to *max_comm_time*. If this condition is fulfilled, this message passing call is a dominating communication statement. Its severity is the fraction of the communication time in relation to the execution time of the *rank_basis*.

3.2.1.7 frequent_communication

```
property frequent_communication (MPIRegion r, Experiment e, Region rank_basis){
LET
  float cost = summary(r,e).sums.comm_time;
IN
  CONDITION:   (typeof(r)==PointToPointPrimitive OR
               (typeof(r)==CollPrimitive AND x.region.type != Barrier)) AND
               cost>0 AND
               cost/summary(r,e).sums.nr_executions<small_messages_threshold;
  CONFIDENCE:  1;
  SEVERITY:    cost/duration(rank_basis,e);
}
```

A communication statement has the property *frequent_communication* if small messages are communicated. The condition compares the execution time per execution with the maximum communication time for small messages. Whether messages are called big depends on the opinion of the tool designer or the application programmer. Therefore, this threshold should be a parameter of the performance tool.
The severity specification is equal to the severity specification of the previous communication properties.

3.2.1.8 big_messages

```
property big_messages (MPIRegion r, Experiment e, Region rank_basis){
LET
  float cost = summary(r,e).sums.comm_time;
  int avrg_length = summary(r,e).sums.message_length/
                    summary(r,e).sums.nr_executions;
IN
  CONDITION:   (typeof(r)==PointToPointPrimitive OR
               (typeof(r)==CollPrimitive AND x.region.type != Barrier)) AND
               cost>0 AND
               avrg_length>big_messages_threshold;
  CONFIDENCE:  1;
  SEVERITY:    cost/duration(rank_basis,e);
}
```

The *big_messages* property is fulfilled by a communication statement if the average message length is greater than a predefined threshold. The severity specification is identical with the severity of previous communication properties.

3.2.1.9 late_sender

```
property late_sender(PointToPointPrimitive r, Experiment e, Region rank_basis){

LET
  float idle_time = summary(r,e).sums.idle_time;
IN
  CONDITION: r.type == Receive AND idle_time>0;
  CONFIDENCE: 1;
  SEVERITY: idle_time/duration(rank_basis,e);
}
```

Only point-to-point receive operations can have the late_sender property. Therefore, the region parameter in the parameter list must be of type *PointToPointPrimitive*.
The condition checks that the statement is receive statement and the idle time is greater than zero. The severity of this property compares the idle time to the duration of *rank_basis*.

3.2.1.10 late_receiver

```
property late_receiver(PointToPointPrimitive r, Experiment e,
                       Region rank_basis){

LET
  float idle_time = summary(r,e).sums.idle_time;
IN
  CONDITION: r.type == Send AND r.semantics == Blocking AND idle_time>0;
  CONFIDENCE: 1;
  SEVERITY: idle_time/duration(rank_basis,e);
}
```

This property can only be proven for send operations that are blocking. Nonblocking operations just setup the transmission and terminate. The severity compares idle time with the duration of *rank_basis*.

3.2.1.11 uneven_mp_distribution

```
property uneven_mp_distribution(MPIRegion r, Experiment e, Region rank_basis){
LET
  float deviation=stdev(sums.duration WHERE sums IN summary(r,e).process_sums);
IN
  CONDITION: (typeof(r)==PointToPointPrimitive OR typeof(r)==CollPrimitive) AND
             deviation > uneven_threshold * summary(r,e).sums.duration/
                         e.nr_processors;
  CONFIDENCE: 1;
  SEVERITY:  summary(r,e).sums.duration/duration(rank_basis,e);
}
```

Any communication statement can have the *uneven_mp_distribution*. The constant *deviation* determines the standard deviation of the execution time of the processes. The condition checks whether the deviation is greater than a threshold multiplied with the mean execution time.

The severity determines the fraction of the execution time in relation to the execution time of *rank_basis*.

3.2.1.12 load_imbalance_at_barrier

```
property load_imbalance_at_barrier(MPIRegion r, Experiment e, Region rank_basis){
LET
  float max_time=max( x.duration WHERE x IN summary(r,e).process_sums );
  float min_time=min( x.duration WHERE x IN summary(r,e).process_sums );
  float max_wait=max_time - min_time;
IN
  CONDITION: (COND1) typeof(r)==CollPrimitive AND
                     r.type==Barrier AND
                     max_wait>0
          || (COND2) typeof(r)==CollPrimitive AND
                     r.type==Barrier AND
                     summary(r,e).sums.idle_time>0;
  CONFIDENCE: 1;
  SEVERITY:  MAX((COND1)->max_wait/(duration(rank_basis,e)/e.nr_processors),
                 (COND2)->summary(r,e).sums.idle_time/duration(rank_basis,e));
}
```

The *load_imbalance_at_barrier* property has two conditions. The first condition can be evaluated if the idle times cannot be measured, while the second condition is based on the idle times. While the confidence value is equal for both conditions, the severity is specified by different formulas. If the first condition is satisfied, the severity is determined by dividing *max_wait* time by the mean duration of each process. If the second condition is fulfilled, the sum of the idle times in all processes is compared to the sum of the individual execution times.

3.2.1.13 slow_slaves

```
property slow_slaves (MPIRegion r, Experiment e, Region rank_basis){

LET
  float idle_time = summary(r,e).sums.idle_time;
IN
  CONDITION: r.paradigm == MasterSlave AND r.role == ReceiveMaster AND
             idle_time>0;
  CONFIDENCE: 1;
  SEVERITY: idle_time/duration(rank_basis,e),
}
```

Both properties, *slow_slaves* and *overloaded_master* described below, are related to the master slave paradigm. In this paradigm, four communication statements are special statements. In the master, a send operation distributes the task to the slaves and a receive operation collects the results. Those statements play the *SendMaster* and *ReceiveMaster* role. In the slaves, a receive operation (*ReceiveSlave* role) accepts tasks and a send operation (*SendSlave* role) returns the results.

The *slow_slaves* property can be proven for the *ReceiveMaster* statement. It identifies a situation where the master waits for results instead of doing useful work.

3.2.1.14 overloaded_master

```
property overloaded_master(MPIRegion r, Experiment e, Region rank_basis){

LET
  float idle_time = summary(r,e).sums.idle_time/(e.nr_processors-1);
IN
  CONDITION: (r.paradigm == MasterSlave AND
              (r.role == ReceiveSlave OR r.role == SendSlave)) AND
             idle_time>0;
  CONFIDENCE: 1;
  SEVERITY: idle_time/duration(rank_basis,e);
}
```

The *overloaded_master* property can be proven for the *ReceiveSlave* and the *SendSlave* operations. If the slaves have to wait for new tasks or for the delivery of the results of finished tasks, the master is too slow.

3.2.2 HPF

This section introduces performance properties of data parallel programs in the context of HPF.
The following performance properties are presented:

- costs
- communication_costs
- forall_synchronization_costs
- io_costs
- parallel_organization_costs
- procedure_remap_costs
- serialization_costs
- uneven_work_distribution
- inspector_cost

3.2.2.1 HPF global definitions

In this section we define a function *summary* returns an object to *HPFRegionSummary*. This object reflects summary information for a specific region and experiment and is used by most HPF properties.

```
HPFRegionSummary summary(HPFRegion r, Experiment e)=
                         UNIQUE({sumr IN e.profile | sumr.region==r});

float duration(Region r, Experiment e)=summary(r,e).sums.duration;
```

Function *duration* denotes the execution time of a region which is the arithmetic mean across all processes that execute the region.
For all HPF performance properties the severity is computed by relating some aspect of the the execution time to the duration of a given *rank_basis* region in the experiment.

3.2.2.2 costs

```
property costs(HPFRegion r, Experiment e, Region rank_basis){
LET
  float cost_sum = summary(r,e).sums.comm_time +
                   summary(r,e).sums.sync_time +
                   summary(r,e).sums.compiler_ovh_time +
                   summary(r,e).sums.io_time;
IN
  CONDITION:   cost_sum>0;
  CONFIDENCE:  1;
  SEVERITY:    cost_sum/duration(rank_basis,e);
}
```

The most general performance property specifies that a region implies some performance costs. The costs of a region can be subdivided into communication, synchronization, compiler overhead, and input/output time. The region accounts for this property if *cost_sum* is greater than 0. The confidence for this condition is one.

The severity of this property is the fraction of the time spent for costs compared to the duration of ranking basis, typically the duration of the main program. Note, that *comm_time*, *sync_time*, *compiler_ovh_time* *io_time*, and *duration* are summary figures across all processes executing the region.

The severity of this property is larger than the severity of the individual properties for each of the categories. This may lead to the selection of the cost property as a performance problem according to the predefined severity threshold while the individual properties, i.e. *communication_costs*, *synchronization_costs*, and *io_costs*, may not be marked as performance problems.

3.2.2.3 communication_costs

```
Property communication_costs (HPFRegion r, Experiment e, Region rank_basis){
LET
  float comm_time = summary(r,e).sums.comm_time;
IN
  CONDITION:   comm_time > 0;
  CONFIDENCE:  1;
  SEVERITY:    comm_time / duration(rank_basis,e);
}
```

This property determines whether a region implies communication. Its condition and severity is based on *comm_time* which is the arithmetic mean across all processes executing region r. The severity is the communication time divided by the execution time of the ranking basis.

3.2.2.4 forall_synchronization_costs

```
Property forall_synchronization_costs (HPFRegion r, Experiment e, Region rank_basis){
LET
  float forall_sync_time = summary(r,e).sums.sync_time;
IN
  CONDITION:   typeof(summary(r,e).region)==HPFLoop AND
               summary(r,e).region.ltype == FORALL AND
               forall_sync_time > 0;
  CONFIDENCE:  1;
  SEVERITY:    forall_sync_time/duration(rank_basis,e);
}
```

FORALL loops may invoke synchronization within and between loop body statements which is specified by property *forall_synchronization_costs*. The severity is the fraction of the synchronization costs at the execution time of the ranking basis.

3.2.2.5 io_costs

```
property io_costs (HPFRegion r, Experiment e, Region rank_basis){
LET
  float io_time = summary(r,e).sums.io_time;
IN
  CONDITION:   io_time > 0;
  CONFIDENCE:  1;
  SEVERITY:    io_time / duration(rank_basis,e);
}
```

Property *io_costs* of a region reflects whether or not any process spends some time in input/output operations. The severity is the fraction of the input/output costs at the execution time of the ranking basis.

3.2.2.6 parallel_organization_costs

```
Property parallel_organization_costs( Region r, Experiment e, Region
rank_basis) {

   CONDITION:  COUNT(procs WHERE procs IN summary(r,e).processes) > 1 ;

   CONFIDENCE: 1;

   SEVERITY:   summary(r,e).compiler_ovh_time / duration(rank_basis,e);
}
```

Execution of a parallel region may be associated with some extra costs implied by a parallelizing compiler. For instance, execution of a statement may be conditional depending on which process is executing the statement. The condition for this property is that a region is executed by more than process. The severity is the time needed to execute the extra code inserted by the compiler.

3.2.2.7 procedure_remap_costs

```
Property procedure_remap_costs (HPFRegion r, Experiment e, Region rank_basis){

  CONDITION:       typeof(summary(r,e).region)==HPFProcedure
             AND (
                   EXISTS
                        pmap  IN summary(r,e).region.decls.format
                   SUCH THAT
                          pmap == PRESCRIPTIVE;
                      OR
                          pmap == DESCRIPTIVE;
             OR
                   EXISTS
                        dir IN summary(r,e).region.alloc
                   SUCH THAT
                        dir == DYNAMIC;
                 )
  CONFIDENCE:  1;
  SEVERITY:    summary(r,e).sums.redistr_time / duration(rank_basis,e);
}
```

Property *procedure_remap_costs* specifies the time spent in remapping arrays in the procedure boundary (remapping of dummy arrays) or body (remapping dynamic arrays). The condition ensures that the region is a procedure. If the procedure has prescriptive or descriptive mapping then remapping may

occur at the procedure boundary. DYNAMIC arrays may be remapped as well. We do not check for remapping caused by a call (to another procedure) inside of the procedure body. The severity is given by the measured redistribution time of arrays of a region divided by the execution time of the ranking basis.

3.2.2.8 serialization_costs

```
Property serialization_costs (HPFRegion r, Experiment seq, Experiment par, Region rank_basis){

LET
  float par_comp_costs = summary(r,par).sums.duration -
                 summary(r,par).sums.comm_time -
                 summary(r,par).sums.sync_time -
                 summary(r,par).sums.idle_time -
                 summary(r,par).sums.compiler_ovh_time -
                 summary(r,par).sums.io_time;

IN
  CONDITION:   par_comp_costs > (duration(r,seq) * loop_serial_threshold)
  CONFIDENCE:  1;
  SEVERITY:    par_comp_costs / (duration(rank_basis,par);
}
```

Property *serialization_costs* reflects whether parallelism has been exploited by a given region. *seq* and *par*, respectively, correspond to a single and multiprocessor experiment.

par_comp_costs defines the parallel computation costs implied by a region which excludes communication, synchronization, idle, compiler overhead, and input/output time. This figure is then compared against the sequential execution time. The severity is given by *par_comp_cost* divided by the parallel execution time of the *rank_basis*.

3.2.2.9 uneven_work_distribution

```
Property uneven_work_distribution (HPFRegion r, Experiment seq, Experiment par,
                                   Region rank_basis) {

LET
  int nr_processes = COUNT(procs WHERE procs IN summary(r,par).processes);
  float opt_duration = duration(r,seq)/nr_processes;

  float deviation = SQRT(SUM (EXP(proc_sum.duration -
                 proc_sum.comm_time -
                 proc_sum.sync_time -
                 proc_sum.idle_time -
                 proc_sum.compiler_ovh_time -
                 proc_sum.io_time - opt_duration, 2)
                   WHERE proc_sum IN summary(r,par).proc_sums ))

IN
  CONDITION: (deviation /  opt_duration) > uneven_threshold
  CONFIDENCE:  1;
  SEVERITY:  summary(r,par).sums.duration / duration(rank_basis,par)
}
```

Property *uneven_work_distribution* specifies how even the computations of a parallel program have been distributed across all processes executing a region. The standard deviation of the computational costs of every process with respect to the optimal duration (sequential execution time divided by number

of processes) is computed. The condition is then given as the variation coefficient compared against a threshold. The severity is defined as the execution time divided by the *rank_basis*.

3.2.2.10 inspector_costs

```
Property inspector_costs (MPIRegion r, Experiment e, Region rank_basis){
LET
  float inspector_time = summary(r,e).sums.inspector_time;
IN
  CONDITION:   inspector_time > 0;
  CONFIDENCE:  1;
  SEVERITY:    inspector_time/duration(rank_basis,e);
}
```

One parallelization strategy for irregular HPF programs implies for each loop (region) a preprocessing (inspector) and an executor phase. The inspector phase – commonly highly execution time intensive – is responsible for the analysis of access patterns and calculation of communication schedules. The executor phase gathers remote data, executes the loop and scatters data to the owning processes. A crucial aspect of this parallelization strategy deals with the problem to reuse the communication schedule of the inspector phase which in many cases is loop invariant.

Property *inspector_costs* denotes the average time spent in the inspector phase across all involved processes. The severity is given by the *inspector_time* divided by the execution time of *rank_basis*.

3.2.3 OpenMP

This section introduces performance properties of shared memory programs. The following performance properties are presented:

- costs
- measurable_costs
- unmeasurable_costs
- non_parallelized_code
- synchronization
- irregular_sync_across_instances
- load_imbalance
- remote_accesses
- remote_access_to_variable
- multiple_transfer_of_same_data
- wrong_page_distribution_for_variable
- parallel_organization

3.2.3.1 OpenMP global definitions

```
SmRegionSummary summary(Region r, Experiment e)=
                    UNIQUE({s IN e.profile WITH s.region==r});

float sync(Region r, Experiment e)=summary(r,e).sums.region_wait +
                                   summary(r,e).sums.region_ctrl +
                                   summary(r,e).sums.cross_thread_dep_wait +
                                   summary(r,e).sums.cross_thread_dep_ctrl ;

float duration(Region r, Experiment e)=summary(r,e).sums.duration;

float remote_access_time(Region r, Experiment e)=
                                   summary(r,e).sums.nr_remote_accesses
                                   * e.system.remote_access_time);
```

The following property specifications make use of those four functions. The *summary* function determines the summary information for a given region and a given experiment.

The *sync* function determines the overhead for synchronization in a given region. It computes the sum of the relevant attributes in the summary class.

The *duration* function returns the execution time of a region. The execution time is determined by the execution time of the master thread in the OpenMP model.

The *remote_access_time* function estimates the overhead for accessing remote memory based on the measured number of accesses and the mean access time of the parallel machine.

3.2.3.2 costs

```
Property costs(Region r, Experiment seq, Experiment par, Region rank_basis){

LET
   float total_costs = duration(r,par) - (duration(r,seq)/par.nr_processors);
IN
   CONDITION: total_costs>0;

   CONFIDENCE: 1;

   SEVERITY: total_costs / duration(rank_basis,par);
}
```

This property specifies that the speedup of the application is not linear. It uses information from two experiments, a sequential run and a parallel run, to compute the costs of parallel execution. Those costs determine the severity of the property.

3.2.3.3 measurable_costs

```
Property measurable_costs(Region r, Experiment e, Region rank_basis){
LET
   float costs = summary(r,e).sums.non_parallelized_code +
                 sync(r,e) +
                 remote_access_time(r,e) +
                 summary(r,e).sums.scheduling +
                 summary(r,e).sums.additional_calc;
IN
   CONDITION: costs>0;

   CONFIDENCE: 1;

   SEVERITY: costs(r,e)/duration(rank_basis,e);
}
```

Performance analysis tools only help in analyzing measurable costs. A region has the *measurable_costs* property if the sum of those costs is greater than zero. The severity of this property is the fraction of those costs relative to the execution time of *rank_basis*.

3.2.3.4 unmeasurable_costs

```
Property unmeasurable_costs(Region r, Experiment seq, Experiment e, Region rank_basis){
LET
   float total_costs = duration(r,e) - (duration(r,seq)/e.nr_processors);
   float costs = summary(r,e).sums.non_parallelized_code +
                 sync(r,e) +
                 remote\_access\_time(r,e) +
                 summary(r,e).sums.scheduling +
                 summary(r,e).sums.additional_calc;
IN
   CONDITION: total_costs-costs>0;

   CONFIDENCE: 1;

   SEVERITY: total_costs(r,e)/duration(rank_basis,e);
}
```

The total cost of the parallel program is the sum of the measurable and the unmeasurable overhead. The *unmeasurable_costs* property determines whether an unmeasurable overhead exists. Its severity is the fraction of this overhead in relation to the execution time of *rank_basis*. If this fraction is high, further tool-supported performance analysis might not be very helpful.

3.2.3.5 non_parallelized_code

```
Property non_parallelized_code(Region r, Experiment e, Region rank_basis){
LET
   float non_parallel_code = summary(r,e).sums.non_parallelized_code>0;
IN
   CONDITION: non_parallel_code>0;

   CONFIDENCE: 1;

   SEVERITY: non_parallel_code/duration(rank_basis,e);
}
```

Non-parallelized code is a very severe problem for application scaling. In the context of analyzing a given program run, its severity is determined in the usual way. If the focus of the analysis is more on application scaling the severity should stress the importance of this property.

3.2.3.6 synchronization

```
Property synchronization(Region r, Experiment e, Region rank_basis){
   CONDITION: sync(r,e)>0;

   CONFIDENCE: 1;

   SEVERITY: sync(r,e)/duration(rank_basis,e);
}
```

A region has the *synchronization* property if any synchronization overhead occurs during its execution. One of the obvious reasons for high synchronization cost is load imbalance.

3.2.3.7 irregular_sync_across_instances

```
Property irregular_sync_across_instances
         (Region r, Experiment e, Region rank_basis){
LET
   float inst_sync(SmInstanceSums sum)=sum.region_wait +
                                       sum.region_ctrl +
                                       sum.cross_thread_dep_wait +
                                       sum.cross_thread_dep_ctrl ;

IN
   CONDITION: stdev(inst_sync(inst_sum)
                    WHERE inst_sum IN summary(r,e).instance_sums)
              > irreg_behaviour_threshold * sync(r,e)/r.nr_executions;

   CONFIDENCE: 1;

   SEVERITY: sync(r,e)/duration(rank_basis,e);
}
```

The *synchronization* property defined above is assigned to regions with synchronization. If the dynamic behaviour of an application changes over the execution time - load imbalance, for example, might occur only in specific phases of the simulation - the whole synchronization overhead might result from specific instances of the region. A region with the *irregular_synch_across_instances* property has an irregular distribution of the synchronization overhead across different instances. The severity is equal to the severity of the synchronization property since the *irregular_synch_across_instances* property is only a more detailed explanation.

3.2.3.8 load_imbalance

```
Property load_imbalance( Region r, Experiment e, Region rank_basis) {

   CONDITION: summary( r, e ).sums.region_wait >0;

   CONFIDENCE: 1;

   SEVERITY: summary( r, e ).sums.region_wait/duration(r,e);
}
```

Work is unevenly distributed to threads in the region. This manifests itself in *region_wait* time. If the *region_wait* time cannot be measured, the property can also be proven based on the execution time of the thread with the longest duration minus the average duration.

3.2.3.9 remote_accesses

```
Property remote_accesses(Region r, Experiment e, Region rank_basis){

   CONDITION: summary(r,e).nr_remote_accesses>0;

   CONFIDENCE: 1;

   SEVERITY: remote_access_time(r,e) / duration(rank_basis,e);

}
```

An important property for code on ccNUMA machines is access to remote memory. Remote memory access implements communication among parallel threads. Since usually only the number of accesses can be measured, the severity is estimated based on the mean access time.

3.2.3.10 remote_access_to_variable

```
Property remote_access_to_variable
   (Region r, Experiment e, String var, Region rank_basis)
{
LET
   VariableRemoteAccesses var_sum =
        UNIQUE({info IN summary(r,e).sums.accessed_variables
               WITH info.var_name==var});
IN
   CONDITION: var_sum.nr_remote_accesses > 0;

   CONFIDENCE: 1;

   SEVERITY:  var_sum.nr_remote_accesses * e.system.remote_access_time
              /duration(rank_basis,e);
}
```

The previous property identifies regions with remote accesses. This property is more specific since its context also includes a specific variable. The property indicates whether accesses to a variable in this region result in remote accesses. It is based on address-range-specific remote access counters, such as the counters provided in the SGI Origin 2000 on page level. The severity of this property is based on the time spent in remote accesses to this variable. Since this property is very useful in explaining a severe remote access overhead for the region, it might be ranked with respect to this region during a more detailed analysis.

3.2.3.11 multiple_transfer_of_same_data

```
Property multiple_transfer_of_same_data
        (Region r, Experiment e, String var, Region rank_basis)
{
LET
   VariableRemoteAccesses var_info =
        UNIQUE({info IN summary(r,e).sums.accessed_variables
               WITH info.var_name==var});
IN
   CONDITION: (Cond1)
                 (var_info.nr_remote_accesses * e.nr_processors > var_info.size
                 / e.system.cache_line_size*summary(r,e).nr_executions)
              OR
              (Cond2)
                 (var_info.nr_remote_accesses * e.nr_processors
                  > 0.5 * var_info.size / e.system.cache_line_size
                    * summary(r,e).nr_executions);

   CONFIDENCE: MAX( (Cond1)->1, (Cond2)->0.5 );

   SEVERITY:  var_info.nr_remote_accesses * e.system.remote_access_time
              /duration(rank_basis,e);
}
```

This property indicates that the same coherence unit, e.g. cache line in ccNUMA-systems, is transferred multiple times between processors. Some of the transfers might be unnecessary.

The conditions in this property compare the number of remote accesses in all threads to the number of cache lines in the data structure. Is the number of remote accesses is larger than the number of cache lines multiplied by the number of executions of that region, i.e. the maximum number of accesses without transferring elements twice, the property is proven with a confidence of one. Is the number of remote accesses only larger than half the potential accesses (Cond2) the confidence is 0.5.

The severity is the same in both cases, i.e. the estimated loss in execution time.

3.2.3.12 wrong_page_distribution_for_variable

```
Property wrong_page_distribution_for_variable
   (Region r, Experiment e, String var, Region rank_basis)
{
LET
   VariableRemoteAccesses var_sum(String var,setof VariableRemoteAccesses info_set) =
        UNIQUE({info IN info_set WITH info.var_name==var});
IN
   CONDITION:
         EXISTS
            thr_sum   IN summary(r,e).thread_sums,
            thr_psum  IN var_sum(var,thr_sum.accessed_variables).page_sums,
            glo_psum  IN {s IN var_sum(var,summary(r,e).accessed_variables).page_sums
                           WITH s.page_no==thr_psum.page_no}
         SUCH THAT
               thr_psum.nr_remote_accesses!=0
            AND
               thr_psum.nr_remote_accesses==glo_psum.nr_remote_accesses;

   CONFIDENCE: 0.5;

   SEVERITY:  var_sum(var,summary(r,e).accesses_variables).nr_remote_accesses
              * e.system.remote_access_time/duration(rank_basis);
}
```

The property identifies a specific reason for remote memory accesses. The condition checks whether remote accesses to at least one of the pages of the variable occur in only a single process. The remote access counts of the other threads for this page have to be zero.

This condition is only an indication for a possibly wrong page distribution. The confidence value is lower than one since it might be possible that another thread executing in the node where the page is allocated accesses this page. If the page would be migrated, remote accesses would then occur for this thread. Since local access counts are not available in that data model, a more precise condition cannot be determined.

The severity of the property depends on the average number of remote accesses in the threads.

3.2.3.13 parallel_organization

```
Property parallel_organization( Region r, Experiment e, Region rank_basis) {

   CONDITION:  typeof(summary(r,e).region)==ParallelRegion
                    AND summary(r, e).nr_executions > 1 ;

   CONFIDENCE: 1;

   SEVERITY:   summary(r,e).nr_executions *
               e.system.parallel_region_cost / duration(rank_basis,e);
}
```

Execution of a parallel region is associated with costs for "going parallel". The severity of these costs depends on the number of instances of the parallel region.

The condition for this property is that the number of executions for a region is greater than one. The severity is the time associated with initiating a parallel regions (sum of *region_ctrl* time) relative to the execution time of the region selected as the ranking basis. It is assumed that the average cost associated with a parallel region (basically the barrier time) is stored as a machine constant (possibly dependent upon p, the number of processors).

Chapter 4

Conclusions and Future Work

This report has described the specification language (ASL) that will be used in the APART working group to describe performance problems in parallel programs. ASL provides constructs to specify performance-related data as an object model, and constructs to describe performance properties, including conditions to prove existence, confidence expressions to support fuzzy information, and severity measures which allows the ranking of performance problems.

The examples presented in this report are data models and performance properties of implementations of three major programming paradigms: MPI, HPF, and OpenMP. Currently, for a specific performance analysis environment on a specific parallel machine, specialized specifications have to be developed since the performance-related data available in the environment must be taken into account. It is one of the goals of APART to collect catalogues of performance properties for the above programming models, so that, in the future, the production of specifications for particular real environments is facilitated.

Three extensions to the current language design will be investigated in the future. Firstly, the current language has no support to find patterns in traces. Some performance problems cannot be proven based on summary information alone. A good example is the message order problem from the grindstone suite, see `www.cs.umd.edu/~hollings`. The messages are sent in reverse order in which they are expected to arrive at the receiver. To check for the presence of this problem, this specific pattern has to be found in the event trace. Either such a pattern can explicitly be described in the language, in a similar way to that allowed in EDL or EARL introduced in Section 1.3, or the pattern can be identified by an external tool and be checked in the specification via a specific external predicate.

Secondly, the language might benefit by being extended to include *template* definitions, which would facilitate the specification of similar performance properties. In the example specifications in this report, some of the properties result directly from the summary information, e.g. io_costs is directly related to the measured time spent in input/output operations. The specifications of those properties are indeed very similar and need not be specified individually.

Thirdly, some sort of meta-properties might be useful. For example, synchronization can be proven based on the summary information, i.e. synchronization exists if the sum, over all processes, of the synchronization time in a region is greater than 0. A more specific property is that associated with individual instances of the region, where some sub-set of the instances are responsible for the synchronization due, for example, to some dynamic changes in the load distribution. Similarly, more specific properties can be sought for other properties as well. Therefore, it would be useful to have some sort of meta-property which evaluates another property in the context of instances instead of the in the context of the entire program run.

Since the data models for the three paradigms do have a common structure, and this common structure will very likely show up in real performance analysis environments, it is covered by a set of base classes that can be reused in new designs. The list of base classes will be extended in the future to cover other common aspects, such as classes representing typical regions, and classes for a standard set of trace events.

Also, in the future, we would like to test the specification techniques in the context of other programming paradigms, such as object oriented programming, distributed applications, multimedia applications, and databases. We would like to verify whether the language is powerful enough to describe the performance

properties found in these environments.

The specification language presented in this report will have to be supplemented by other specification notations. For example, a notation is required to describe the data supplied by existing analysis tools in the target performance analysis environment. Also required is a notation to provide the means for specifying the analysis process of automated performance analysis tools. Both specifications have been introduced in Section 1.2. Besides the coordinated design of these languages, the issue of the efficient translation of the specifications into data and/or code for use by automated analysis tools needs to be investigated. We anticipate that this latter topic might well lead to a design for all the languages that is tailored more towards tool implementation.

Bibliography

[AhSeUl 88] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools.* Series in Computer Science. Addison Wesley, 1988.

[BaWi 83] P. Bates, J.C. Wileden: *High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach*, The Journal of Systems and Software, Vol. 3, pp. 255-264, 1983

[BeGeKr 96] R. Berrendorf, M. Gerndt, A. Krumme: *A Programming Environment for Parallel Computers with Global Address Space*, Workshop on High-Level Programming Models and Supportive Environments (HIPS '96), in combination with IPPS '96, IEEE, pp. 10-16, 1996

[CuSiGu 99] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach* . Morgan Kaufmann Publisher Inc., 1999.

[DaMe 98] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, January/March 1998.

[EsMaLu 98] A. Espinosa, T. Margalef, E. Luque: *Automatic Performance Evaluation of Parallel Programs*, Sixth Euromicro Workshop on Parallel and Distribued Processing, 1998

[FaScPa 99] Thomas Fahringer, Bernhard Scholz, and Mario Pantano. Execution-Driven Performance Analysis for Distributed and Parallel Systems. Institute for Software Technology and Parallel Systems, University of Vienna, Liechtensteinstr. 22, A-1090 Wien. Technical Report, June, 1999.

[Fa 95] T. Fahringer. Estimating and Optimizing Performance for Parallel Programs. IEEE Computer, 28(11), pp. 47-56, Nov. 1995.

[Fa 96] T. Fahringer. Automatic Performance Prediction of Parallel Programs. Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March, 1996.

[GeKr 97] M. Gerndt, A. Krumme: *A Rule-based Approach for Automatic Bottleneck Detection in Programs on Shared Virtual Memory Systems*, Second Workshop on High-Level Programming Models and Supportive Environments (HIPS '97), in combination with IPPS '97, IEEE, 1997

[GeKrOz 95] M. Gerndt, A. Krumme, S. Özmen: *Performance Analysis for SVM-Fortran with OPAL*, Proceedings Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'95), Athens, Georgia, pp. 561-570, 1995

[HeMa 95] B.R. Helm, A.D. Malony: *Automating Performance Diagnosis: a Theory and Architecture*, International Workshop on Computer Performance Measurement and Analysis (PERMEAN '95), 1995

[HPF 93] High Performance FORTRAN Language Specification. Technical Report, Version 2.0.δ, Rice University, Houston, TX, October 1996.

[KaLeObWa 98] Hockauf, R.; Karl, W.; Leberecht, M.; Oberhuber, M.; Wagner, M.: Exploiting Spatial and Temporal Locality of Accesses: A New Hardware-Based Monitoring Approach for DSM Systems. In: D. Pritchard, Jeff Reeve (Eds.): Euro-Par'98 Parallel Processing / 4th International Euro-Par Conference Southampton, UK, September 1-4, 1998 Proceedings. Springer-Verlag, Heidelberg, Lecture Notes in Computer Science Vol.1470, 1998, pp. 206-215

[MCCHI 95] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall: *The Paradyn Parallel Performance Measurement Tool*, IEEE Computer, Vol. 28, No. 11, pp. 37-46, 1995

[GaMo 98] J. Galarowicz, B. Mohr: *Analyzing Message Passing Programs on the Cray T3E with PAT and VAMPIR*, Fourth European CRAY-SGI MPP Workshop Garching/München, Research Centre Juelich Technical Report, FZJ-ZAM-IB-9809, 1998

[MoBrMa 94] B. Mohr and D. Brown and A. Malony. TAU: A portable parallel program analysis environment for pC++. CONPAR, Linz, Austria, 1994.

[NaArWeHoSo 96] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer 12(1), pp. 69-80, 1996.

[Paradyn 98] Paradyn Project: *Paradyn Parallel Performance Tools: User's Guide*, Paradyn Project, University of Wisconsin Madison, Computer Sciences Department, 1998

[Mu 99] Nandini Mukhopadhyay (Mukherjee). On the effectiveness of feedback-guided parallelisation. PhD Thesis, University of Manchester, Department of Computer Science, September, 1999.

[Mu 00] N. Mukherjee, G.D. Riley and J.R. Gurd. FINESSE: A Prototype Feedback-guided Performance Enhancement System, Accepted for PDP2000, to be held in Rhodes in January 2000.

[RuJaBo 99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison Wesley Longman, Reading, Mass., 1999.

[SnOtHUWaDo 98] M. Snir, St. Otto, St. Huss-Lederman, D. Walker, J. Dongarra: *MPI - The Complete Reference*, MIT Press, ISBN 0-262-69216-3, 1998

[WoMo 98] F. Wolf, B. Mohr: *EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs*, 7th International Conference on High-Performance Computing and Networking (HPCN'99), A. Hoekstra, B. Hertzberger (Eds.), Lecture Notes in Computer Science, Vol. 1593, pp. 503-512, 1999

Appendix A

Unified Modeling Language Class Diagrams

The object models are presented as Unified Modelling Language (UML) class diagrams [RuJaBo 99],

$$http://www.rational.com/uml$$

and as textual property specification documents. Figure A.1 shows a simple UML class diagrams. The boxes represent classes, the class name is shown in the upper part and the class's attributes in the lower part. The attributes are represented by its name followed by its type. Closed arrows represent the specialization of generalization relationship. For example, SaloonCar and Caravan are a specializations or subclasses of Car.

UML diagrams provide two other types of relationships: associations and aggregations. Associations are represented by plain lines and open arcs. In contrast to a plain line, an arc defines that the association in navigable in this direction. The name for accessing the associated object is written near to this object, in the diagrams in Figure A.1 the potential drivers of a car are identified via Drivers. In addition, the cardinality of the associated objects can be specified at the end points of an association or aggregation. For example, each car can have 1 or more potential drivers and each person can drive 0 or more cars.

Aggregations are represented as lines or arcs with a diamond near to the aggregate class. An aggregation is a *has a*-relationship. The diagrams in Figure A.1 specifies that a car has four wheels. An aggregation relationship is used instead of an association if the destruction of the aggregate object also leads to the destruction of the attribute objects.

The UML is a very powerful specification language. The above paragraphs only explain those feature that are used in the performance model diagrams. When designing and implementing software, UML

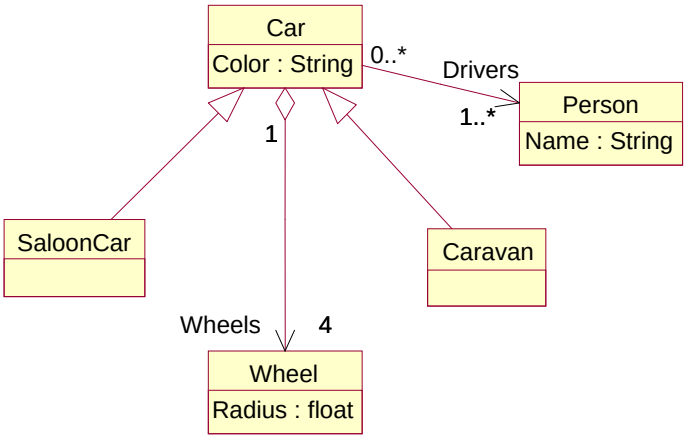

Figure A.1: Object model in the Unified Modelling Language (UML)

specification are automatically translated into appropriate programming language constructs, such as classes in Java or C++ and SQL specifications for relational databases. In a similar procedure the diagrams of performance-related data models can be transferred into the APART specification language.

Appendix B

APART Base Class Library

```
//*********************************************************************
//
// APART Base Classes
//
//*********************************************************************

class Application {
    String name;                    //Name of application
    setof Version versions;         //Versions

}


class Version {
    int version_no;                 //Version number
    setof SourceFile files;         //Source files
    setof Experiment experiments;   //Multiple experiments

}

// Classes SourceFile and Region model pure static information

class SourceFile {
    String name;                    //File name
    String contents;                //File contents
    setof Region regions;           //Regions included in file

}

class Region {
    Position start_pos;             //Start position (line, column) in file
    Position end_pos;               //End position
    setof Region sub_regions;       //Regions nested in that region
    setof Region successors;        //Successor regions according to region
    setof Region predecessors;      //Predecessor regions according to region
                                    //control flow
}

//The following classes are used to model dynamic performance
//information. It includes the summary data per region
//(entire run, all processes) and the traces of the processes.
```

```
class Experiment {
    DateTime start_time;           //Start time of the experiment
    int nr_processors;             //Number of used processors
    setof RegionSummary profile;   //Summed up information for events
    setof Event trace;             //Trace records for individual events
    Machine system;                //Machine chracterization
}

class Event {
    float timestamp;               //Timestamp of event
    Process process_id;            //Process number
}

class RegionSummary {
    Region region;                 //Region with measured data
}


//The following classes are utility classes for the different
//programming paradigms.

class Dependence {
    Region src;                    //Source of dependence
    Region dst;                    //Destination of dependence
    GeneralTypes.dep_type type;    //Type of data dependence
    GeneralTypes.dep_dir direction;//Direction of dependence
    int distance;                  //Distance of dependence
    int level;                     //Loop level that carries dependence
}

class GeneralTypes{
  enum dep_dir { '<', '>', '=');   //Direction of dependence
  enum dep_type{ True, Anti, Output); //Type of dependence
}

class LoopHeader {                 //Describes loop bounds
    String lower;                  //Lower bound
    String upper;                  //Upper bound
    String stride;                 //Stride of loop
}

class Position {
    int line;                      //Line in file
    int col;                       //Column in that line

}

class Process {
    int process_id;                //Process number

}

class Machine {                    //Generic parallel machine
    int nr_processors;             //All physical processors
}
```

Appendix C

MPI Property Specification

```
//*********************************************************************
// APART Example Property Specification
//
// Message Passing Paradigm Performance Data Model
//
//*********************************************************************

PERFORMANCE DATA

//*********************************************************************
//MPI static information
//*********************************************************************

class MPIRegion extends Region {
   MPITypes.ParadigmType paradigm;    //defines implemented paradigm
   MPITypes.ParadigmRole role;        //role of this region in paradigm
}

class LoopRegion extends MPIRegion {
   LoopType Type;                     //e.g. do, while, forall
}

class PointToPointPrimitive extends MPIRegion {
   MPITypes.CommType type;            //e.g. send, receive
   MPITypes.CommMode mode;            //buffered, synchronous, ready
   MPITypes.CommSemantics semantics;  //blocking, nonblocking

}

class CollPrimitive extends MPIRegion {
   MPITypes.CollType type;            //reduction, broadcast
   MPITypes.CollSyncType sync;        //barrier, nobarrier

}

class MPIio extends MPIRegion {       //Class models MPIio routines
}

//*********************************************************************
//MPI dynamic information: Summary data
//*********************************************************************
```

```
class MPIRegionSummary extends RegionSummary {
    MPISummary sums;                  //Summary data all processes
    setof MPIProcessSummary process_sums; //Summary data per process
}


class MPISummary {
   float duration;                    //Execution time sum for processes
   float comm_time;                   //Communication time sum
   float io_time;                     //IO time sum
   float sync_time;                   //Barrier synchronization time
   float idle_time;                   //Idle time sum
   int message_length;                //Sum of the length of all messages
                                      //sent during execution of that region
   int nr_executions;                 //Number of executions
}

class MPIProcessSummary {             //Summary data per process
    Process process_id;               //Process number
    float comm_time;
    float io_time;
    float sync_time;
    float idle_time;
    int message_length;
    float duration;
    float nr_executions;
}

//*********************************************************************
//MPI dynamic information: Events
//*********************************************************************

class PointToPointEvent extends Event {
   PointToPointPrimitive statement;    //Executed MPI routine
   CommInfo time_break_down;           //Detailed timings of that execution
}


class CollEvent extends Event {
   CollPrimitive statement;            //Executed MPI routine
   Communicator context;               //Active communicator
   CommInfo time_break_down;           //Detailed timings of that execution
}

class CommInfo {                       //Detailed timings per communication
   float duration;                     //Total execution time
                                       // =idle_time + transmission_time
   float transmission_time;            //Time for transferring data
   float idle_time;                    //Waiting time
   Process partner;                    //Communication target
   int message_length;                 //Message length
}


class Communicator {
    setof Process processes;
```

```
}


//*********************************************************************
//MPI utility classes
//*********************************************************************

class MPITypes {
   enum LoopType {Do, While, Forall};
   enum CommMode {Buffered, Sync, Ready};
   enum CollType {Reduction, Broadcast};
   enum CommType {Send, Recv, SendRecv};
   enum CommSemantics {Blocking, Nonblocking};
   enum CollSyncType {Barrier, Nobarrier};
   enum ParadigmType {MasterSlave, DivideConquer, Farming};
   enum ParadigmRole {MasterSend, MasterRecv, SlaveSend, SlaveRecv};
}

//*********************************************************************
// APART Example Property Specification
//
// Message Passing Paradigm Performance Properties
//
//*********************************************************************

PERFORMANCE PROPERTIES

LET

MPIRegionSummary summary(MPIRegion r, Experiment e)=
                     UNIQUE({sumr IN e.profile WITH sumr.region==r});

float duration(MPIRegion r, Experiment e)=summary(r,e).sums.duration;

IN

property costs(MPIRegion r, Experiment e, Region rank_basis){
LET
  float CostSum = summary(r,e).sums.comm_time +
                  summary(r,e).sums.sync_time +
                  summary(r,e).sums.io_time;
IN
  CONDITION:   CostSum>0;
  CONFIDENCE:  1;
  SEVERITY:    CostSum/duration(rank_basis,e);
}

property communication_costs (MPIRegion r, Experiment e, Region rank_basis){
LET
  float cost = summary(r,e).sums.comm_time;
IN
  CONDITION:   cost>0;
  CONFIDENCE:  1;
  SEVERITY:    cost/duration(rank_basis,e);
}
```

```
property synchronization_costs (Region r, Experiment e, Region rank_basis){

LET
  float barrier_time = summary(r,e).sums.sync_time;
IN
  CONDITION:   barrier_time > 0;
  CONFIDENCE: 1;
  SEVERITY:   barrier_time/duration(rank_basis,e)
}


property io_costs (Region r, Experiment e, Region rank_basis){

LET
  float io_time = summary(r,e).sums.io_time;
IN
  CONDITION:   io_time > 0;
  CONFIDENCE: 1;
  SEVERITY:   io_time/duration(rank_basis,e)
}


property dominating_communication(MPIRegion r, Experiment e, Region rank_basis){
LET
  setof MPIRegionSummary comm_summaries=
           {x IN e.profile
            WITH
              typeof(x.region)==PointToPointPrimitive
            OR
              (typeof(x.region)==CollPrimitive AND x.region.type != Barrier)
           };
  float max_comm_time = MAX(sum.sums.duration WHERE sum IN CommSummaries);
IN
  CONDITION: (typeof(r)==PointToPointPrimitive OR
              (typeof(r)==CollPrimitive AND x.region.type != Barrier)) AND
              summary(r,e).sums.duration==max_comm_time;
  CONFIDENCE: 1;
  SEVERITY: max_comm_time/duration(rank_basis,e);
}


property frequent_communication (MPIRegion r, Experiment e, Region rank_basis){
LET
  float cost = summary(r,e).sums.comm_time;
IN
  CONDITION:   (typeof(r)==PointToPointPrimitive OR
                (typeof(r)==CollPrimitive AND x.region.type != Barrier)) AND
                cost>0 AND
                cost/summary(r,e).sums.nr_executions<small_messages_threshold;
  CONFIDENCE:  1;
  SEVERITY:    cost/duration(rank_basis,e);
}


property big_messages (MPIRegion r, Experiment e, Region rank_basis){
LET
  float cost = summary(r,e).sums.comm_time;
```

```
  int avrg_length = summary(r,e).sums.message_length/
                    summary(r,e).sums.nr_executions;
IN
  CONDITION:   (typeof(r)==PointToPointPrimitive OR
               (typeof(r)==CollPrimitive AND x.region.type != Barrier)) AND
               cost>0 AND
               avrg_length>big_messages_threshold;
  CONFIDENCE:  1;
  SEVERITY:    cost/duration(rank_basis,e);
}


property late_sender(PointToPointPrimitive r, Experiment e, Region rank_basis){

LET
  float idle_time = summary(r,e).sums.idle_time;
IN
  CONDITION: r.type == Receive AND idle_time>0;
  CONFIDENCE: 1;
  SEVERITY: idle_time/duration(rank_basis,e);
}


property late_receiver(PointToPointPrimitive r, Experiment e,
                       Region rank_basis){

LET
  float idle_time = summary(r,e).sums.idle_time;
IN
  CONDITION: r.type == Send AND r.semantics == Blocking AND idle_time>0;
  CONFIDENCE: 1;
  SEVERITY: idle_time/duration(rank_basis,e);
}


property uneven_mp_distribution(MPIRegion r, Experiment e, Region rank_basis){
LET
  float deviation=stdev(sums.duration WHERE sums IN summary(r,e).process_sums);
IN
  CONDITION: (typeof(r)==PointToPointPrimitive OR typeof(r)==CollPrimitive) AND
             deviation > uneven_threshold * summary(r,e).sums.duration/
                         e.nr_processors;
  CONFIDENCE: 1;
  SEVERITY:  summary(r,e).sums.duration/duration(rank_basis,e);
}


property load_imbalance_at_barrier(MPIRegion r, Experiment e, Region rank_basis){
LET
  float max_time=max( x.duration WHERE x IN summary(r,e).process_sums );
  float min_time=min( x.duration WHERE x IN summary(r,e).process_sums );
  float max_wait=max_time - min_time;
IN
  CONDITION: (COND1) typeof(r)==CollPrimitive AND
                     r.type==Barrier AND
                     max_wait>0
          || (COND2) typeof(r)==CollPrimitive AND
```

```
                    r.type==Barrier AND
                    summary(r,e).sums.idle_time>0;
  CONFIDENCE: 1;
  SEVERITY:  MAX((COND1)->max_wait/(duration(rank_basis,e)/e.nr_processors),
                 (COND2)->summary(r,e).sums.idle_time/duration(rank_basis,e));
}


property slow_slaves (MPIRegion r, Experiment e, Region rank_basis){

LET
  float idle_time = summary(r,e).sums.idle_time;
IN
  CONDITION: r.role == ReceiveMaster and idle_time>0;
  CONFIDENCE: 1;
  SEVERITY: idle_time/duration(rank_basis,e),
}


property overloaded_master(MPIRegion r, Experiment e, Region rank_basis){

LET
  float idle_time = summary(r,e).sums.idle_time/(e.nr_processors-1);
IN
  CONDITION: (r.role == ReceiveSlave OR r.role == SendSlave) AND idle_time>0;
  CONFIDENCE: 1;
  SEVERITY: idle_time/duration(rank_basis,e);
}


END
```

Appendix D

HPF Property Specification

```
//*********************************************************************
// APART Example Property Specification
//
// Data Parallel Programming Paradigm Performance Data Model
//
//*********************************************************************

PERFORMANCE DATA


//*********************************************************************
//HPF static information
//*********************************************************************

class HPFRegion extends Region {
    setof Dependence deps;              // data dependence information of this region
    setof HPFDirective dirs;            // HPF directives
    setof HPFDataDeclaration decls;     // HPF declarations
}

//Examples for HPF regions

class HPFProcedure extends HPFRegion {  // Procedure is function or subroutine
}

class HPFLoop extends HPFRegion {
    HPFLoopType ltype;                  // type of loop
}

class HPFIfBlock extends HPFRegion {

}

class HPFBasicBlock extends HPFRegion {

}

class HPFProcedureCall extends HPFRegion {

}
```

```
class HPFArrayAssignment extends HPFRegion {

}


//*********************************************************************
//HPF dynamic information: summary data
//*********************************************************************

class HPFRegionSummary extends RegionSummary {
    setof Process processes;                    // set of processes executing this region
    HPFSummary sums;                            // performance summary across all processes
    setof HPFProcessSummary process_sums;       // performance summary per process
}

class HPFSummary {
         // summary information (arithmetic mean) across all
         // processes for a given region

    int nr_executions;
        // average number of times this region has been executed
        // across all processes executing this region
    float duration;             // execution time
    float comm_time;            // communication time
    float dep_comm_time;        // communication time caused by data dependences
    float align_comm_time;      // communication time caused by data alignment
    float sync_time;            // barrier, reduce, allreduce, ...
    float idle_time;            // idle time
    float io_time;              // input/output time
    float compiler_ovh_time;    // compiler overhead time
    float inspector_time;       // time for inspector phase
    float redistr_time;         // time for redistribution of data structures
    int nr_cache_misses;        // number of cache misses
}

class HPFProcessSummary {
         // performance summary (arithmetic mean) for
         // individual process across all region instances

    Process process;            // process identification

    int nr_executions;
        // number of times this region has been executed by process
    float duration;             // execution time
    float comm_time;            // communication time
    float dep_comm_time;        // communication time caused by data dependences
    float align_comm_time;      // communication time caused by data alignment
    float sync_time;            // barrier, reduce, allreduce, ...
    float idle_time;            // idle time
    float io_time;              // input/output time
    float compiler_ovh_time;    // compiler overhead time
    float inspector_time;       // time for inspector phase
    float redistr_time;         // time for redistribution of data structures
    int nr_cache_misses;        // number of cache misses
}

//*********************************************************************
```

```
//HPF untility classes
//*********************************************************************

class HPFTypes {
    enum hpf_directive { PROCESSORS, DISTRIBUTE, ALIGN, RESHAPE, INDEPENDENT,DYNAMIC,...};
    enum hpf_loop_type { DO, INDEPENDENT, FORALL,...};
    enum hpf_var_arr {VARIABLE, ARRAY};
    enum hpf_distr_type {BLOCK, CYCLIC, "*", ":"}
    enum hpf_distr_format {PRESCRIPTIVE, DESCRIPTIVE, TRANSCRIPTIVE, INHERIT}
    enum hpf_alloc {DYNAMIC, STATIC};
}


class HPFDataDeclaration {
    String name;                        // name of data
    String data_type;                   // type of data (int, float, ...)
    int rank;                           // rank of data
    HPFTypes.hpf_var_arr type;          // type of data (variable or array)
    HPFTypes.hpf_alloc alloc;           // allocation type
    HPFTypes.hpf_distr_format format;   // prescriptive, descriptive, transcriptive,
                                        // inherit distribution format
    setof ArrayDimension dims;          // more information for each dimension of arrays
}

class ArrayDimension {
    HPFDataDeclaration decl;            // declaration of associated array
    int size;                           // size of dimension
    setof Process processes;            // set of processes onto which dimension is mapped
    HPFTypes.hpf_distr_type type;       // data distribution type
    int block_size;                     // for CYCLIC(block_size) or BLOCK(block_size)
    ArrayDimension align;               // aligned with some other array dimension
}

class HPFDirective {
    HPFTypes.hpf_directive hpf_dir;     // HPF directives
}

class HPFLoopType {
    HPFTypes.hpf_loop_type ltype;       // HPF loop type
}




//*********************************************************************
// APART Example Property Specification
//
// Data Parallel Programming Paradigm Performance Properties
//
//*********************************************************************



Property communication_costs (HPFRegion r, Experiment e, Region rank_basis){
LET
  float comm_time = summary(r,e).sums.comm_time;
IN
```

```
  CONDITION:   comm_time > 0;
  CONFIDENCE:  1;
  SEVERITY:    comm_time / duration(rank_basis,e);
}


Property forall_synchronization_costs (HPFRegion r, Experiment e, Region rank_basis){
LET
  float forall_sync_time = summary(r,e).sums.sync_time;
IN
  CONDITION:   typeof(summary(r,e).region)==HPFLoop AND
               summary(r,e).region.ltype == FORALL AND
               forall_sync_time > 0;
  CONFIDENCE:  1;
  SEVERITY:    forall_sync_time/duration(rank_basis,e);
}


property io_costs (HPFRegion r, Experiment e, Region rank_basis){
LET
  float io_time = summary(r,e).sums.io_time;
IN
  CONDITION:   io_time > 0;
  CONFIDENCE:  1;
  SEVERITY:    io_time / duration(rank_basis,e);
}


Property parallel_organization_costs( Region r, Experiment e, Region
rank_basis) {

   CONDITION:  COUNT(procs WHERE procs IN summary(r,e).processes) > 1 ;

   CONFIDENCE: 1;

   SEVERITY:   summary(r,e).compiler_ovh_time / duration(rank_basis,e);
}


Property procedure_remap_costs (HPFRegion r, Experiment e, Region rank_basis){

  CONDITION:       typeof(summary(r,e).region)==HPFProcedure
             AND (
                   EXISTS
                        pmap  IN summary(r,e).region.decls.format
                   SUCH THAT
                          pmap == PRESCRIPTIVE;
                      OR
                          pmap == DESCRIPTIVE;
             OR
                   EXISTS
                        dir IN summary(r,e).region.alloc
                   SUCH THAT
                        dir == DYNAMIC;
                )
  CONFIDENCE:  1;
  SEVERITY:    summary(r,e).sums.redistr_time / duration(rank_basis,e);
```

```
}


Property procedure_remap_costs (HPFRegion r, Experiment e, Region rank_basis){

  CONDITION:       typeof(summary(r,e).region)==HPFProcedure
             AND (
                   EXISTS
                        pmap  IN summary(r,e).region.decls.format
                   SUCH THAT
                          pmap == PRESCRIPTIVE;
                      OR
                          pmap == DESCRIPTIVE;
             OR
                   EXISTS
                        dir IN summary(r,e).region.alloc
                   SUCH THAT
                        dir == DYNAMIC;
                 )
  CONFIDENCE:  1;
  SEVERITY:    summary(r,e).sums.redistr_time / duration(rank_basis,e);
}

Property serialization_costs (HPFRegion r, Experiment e, Region rank_basis){

LET
  float par_comp_costs = summary(r,e).sums.duration -
                  summary(r,e).sums.comm_time -
                  summary(r,e).sums.sync_time -
                  summary(r,e).sums.idle_time -
                  summary(r,e).sums.compiler_ovh_time -
                  summary(r,e).sums.io_time;

IN
  CONDITION:   par_comp_costs > (duration(r,seq) * loop_serial_threshold)
  CONFIDENCE:  1;
  SEVERITY:    par_comp_costs / (duration(rank_basis,e);
}



Property uneven_work_distribution (HPFRegion r, Experiment e, Region rank_basis){

LET
  int nr_processes = COUNT(procs WHERE procs IN summary(r,e).processes);
  float arith_mean = duration(r,seq)/nr_processes;

  float deviation = SQRT(summary(r,e).sums.duration -
                  summary(r,e).sums.comm_time -
                  summary(r,e).sums.sync_time -
                  summary(r,e).sums.idle_time -
                  summary(r,e).sums.compiler_ovh_time -
                  summary(r,e).sums.io_time - arith_mean)^2);
IN
  CONDITION: (deviation /  arith_mean) > uneven_threshold
  CONFIDENCE:  1;
  SEVERITY:  summary(r,e).sums.duration / duration(rank_basis,e)
```

```
}


Property inspector_costs (MPIRegion r, Experiment e, Region rank_basis){
LET
  float inspector_time = summary(r,e).sums.inspector_time;
IN
  CONDITION:   inspector_time > 0;
  CONFIDENCE:  1;
  SEVERITY:    inspector_time/duration(rank_basis,e);
}
```

Appendix E

OpenMP Property Specification

```
//*********************************************************************
// APART Example Property Specification
//
// Shared Memory Paradigm Performance Data Model
//
//*********************************************************************


PERFORMANCE DATA

//*********************************************************************
//SMP static information
//*********************************************************************

class SmRegion extends Region {         //Region subclass for SM regions
    Dependence deps[];                  //Dependence information

}


//Sequential regions

class SequentialRegion extends SmRegion {
                                        //Common information for
                                        // sequential regions go here
}

//Examples for sequential regions

class FunctionCall extends SequentialRegion {

}


class Function extends SequentialRegion {

}


class IfBlock extends SequentialRegion {
```

```
}


class BasicBlock extends SequentialRegion {

}

//Parallel regions

class ParallelRegion extends SmRegion {
                                        //Common information for
                                        // sequential regions go here
   boolean no_wait_exit;                //Region is not terminated by barrier
}

//Examples for parallel regions

class PDo extends ParallelRegion {
    SMTypes.scheduling_type scheduling_strategy;
                                        //Static, Dynamic, Guided ...
}
class PSection extends ParallelRegion {

}
class PRegion extends ParallelRegion {

}


class SMTypes {
   enum scheduling_type {Static, Dynamic, Guided};
}


class DSMMachine extends Machine {      //Properties of target machine
    int remote_access_time;             //Remote access time
}

//*********************************************************************
//SMP dynamic information: summary data
//*********************************************************************

class SmRegionSummary extends RegionSummary {
                                       //Summary information for region
   SmSums sums;                        //Sums for whole execution
   setof SmInstanceSums instance_sums; //Sums per region instance
   int nr_executions                   //Number of instances
}

class SmSums {
   float duration;                     //Execution time of master
   float non_parallelized_code;        //Sequential time (duration -
                                       // duration for parallel regions)
   fload seq_fraction;                 //Seq_time / duration
   int nr_remote_accesses;             //Number of remote memory accesses
   float scheduling;                   //Compiler and or user scheduling time
   float additional_calc;              //Time for additional calculations
```

```
   float cross_thread_dep_crtl;        //Synchronization except entry and exit
                                       // barrier and except waiting time
                                       // in locks etc.
   float cross_thread_dep_wait;        //Synchronization waiting time except
                                       // waiting in entry or exit barrier
   float region_wait;                  //Waiting in entry or exit barrier
   float region_ctrl;                  //Time for instructions at master
                                       // e.g. barriers and organization
   int nr_cache_misses;                //Number of cache misses
   setof SmThreadSums thread_sums;     //Thread specific summary data
   setof VariableRemoteAccesses
       accessed_variables;             //These objects determine the number of
                                       // remote accesses for variables
                                       // accessed in that region
}


class SmThreadSums {
    int thread_no;                     //Thread id
    float region_wait;
    int nr_remote_accesses;
    float additional_calc;
    float cross_thread_dep_crtl;
    float cross_thread_dep_wait;
    int nr_cache_misses;
    setof VariableRemoteAccesses
       accessed_variables;             //These objects determine the number of
                                       // remote accesses for variables
                                       // accessed in that region
}


class SmInstanceSums {
   int nr_threads;                     //Number of threads executing the region
   float duration;                     //Execution time of master
   float non_parallelized_code;
   fload seq_fraction;
   int nr_remote_accesses;
   float scheduling;
   float additional_calc;
   float cross_thread_dep_crtl;
   float cross_thread_dep_wait;
   float region_wait;
   float region_ctrl;
   int nr_cache_misses;
   setof SmThreadInstanceSums thread_sums;
                                       //Thread specific instance information
   setof VariableRemoteAccesses
       accessed_variables;             //These objects determine the number of
                                       // remote accesses for variables
                                       // accessed in that region

}


class SmThreadInstanceSums {
    int thread_no;                     //Thread id
```

```
    float region_wait;
    int nr_remote_accesses;
    float additional_calc;
    float cross_thread_dep_crtl;
    float cross_thread_dep_wait;
    int nr_cache_misses;
    setof VariableRemoteAccesses
       accessed_variables;            //These objects determine the number of
                                      // remote accesses for variables
                                      // accessed in that region
}


class VariableRemoteAccesses {
    String var_name;                  //Name of a variable accessed in region
    int nr_remote_accesses;           //Number of remote accesses via
                                      // references to this variable
    int size;                         //Size in bytes
    setof PageRemoteAccesses;         //For each page of this variable the
                                      // number of remote accesses
}


class PageRemoteAccesses{
    int page_no;                      //Page number related to
                                      // virtual address space
    int nr_remote_accesses;           //Number of remote accesses
}


//*********************************************************************
// APART Example Property Specification
//
// Shared Memory Paradigm Performance Properties
//
//*********************************************************************


PERFORMANCE PROPERTIES
LET

SmRegionSummary summary(Region r, Experiment e)=
                  UNIQUE({s IN e.profile WITH s.region==r});

float sync(Region r, Experiment e)=summary(r,e).sums.region_wait +
                                   summary(r,e).sums.region_ctrl +
                                   summary(r,e).sums.cross_thread_dep_wait +
                                   summary(r,e).sums.cross_thread_dep_ctrl ;

float duration(Region r, Experiment e)=summary(r,e).sums.duration;

float remote_access_time(Region r, Experiment e)=
                                   summary(r,e).sums.nr_remote_accesses
                                   * e.system.remote_access_time);

IN
```

```
Property costs(Region r, Experiment seq, Experiment par, Region rank_basis){

LET
   float total_costs = duration(r,par) - (duration(r,seq)/par.nr_processors);
IN
   CONDITION: total_costs>0;

   CONFIDENCE: 1;

   SEVERITY: total_costs / duration(rank_basis,par);
}


Property measurable_costs(Region r, Experiment e, Region rank_basis){
LET
   float costs = summary(r,e).sums.non_parallelized_code +
                 sync(r,e) +
                 remote_access_time(r,e) +
                 summary(r,e).sums.scheduling +
                 summary(r,e).sums.additional_calc;
IN
   CONDITION: costs>0;

   CONFIDENCE: 1;

   SEVERITY: costs(r,e)/duration(rank_basis,e);
}


Property unmeasurable_costs(Region r, Experiment seq, Experiment e, Region rank_basis){
LET
   float total_costs = duration(r,e) - (duration(r,seq)/e.nr_processors);
   float costs = summary(r,e).sums.non_parallelized_code +
                 sync(r,e) +
                 remote\_access\_time(r,e) +
                 summary(r,e).sums.scheduling +
                 summary(r,e).sums.additional_calc;
IN
   CONDITION: total_costs-costs>0;

   CONFIDENCE: 1;

   SEVERITY: total_costs(r,e)/duration(rank_basis,e);
}


Property non_parallelized_code(Region r, Experiment e, Region rank_basis){
LET
   float non_parallel_code = summary(r,e).sums.non_parallelized_code>0;
IN
   CONDITION: non_parallel_code>0;

   CONFIDENCE: 1;

   SEVERITY: non_parallel_code/duration(rank_basis,e);
}
```

```
Property synchronization(Region r, Experiment e, Region rank_basis){
   CONDITION: sync(r,e)>0;

   CONFIDENCE: 1;

   SEVERITY: sync(r,e)/duration(rank_basis,e);
}


Property irregular_sync_across_instances
         (Region r, Experiment e, Region rank_basis){
LET
   float inst_sync(SmInstanceSums sum)=sum.region_wait +
                                       sum.region_ctrl +
                                       sum.cross_thread_dep_wait +
                                       sum.cross_thread_dep_ctrl ;

IN
   CONDITION: stdev(inst_sync(inst_sum)
                    WHERE inst_sum IN summary(r,e).instance_sums)
              > irreg_behaviour_threshold * sync(r,e)/r.nr_executions;

   CONFIDENCE: 1;

   SEVERITY: sync(r,e)/duration(rank_basis,e);
}


Property load_imbalance( Region r, Experiment e, Region rank_basis) {

   CONDITION: summary( r, e ).sums.region_wait >0;

   CONFIDENCE: 1;

   SEVERITY: summary( r, e ).sums.region_wait/duration(r,e);
}


Property remote_accesses(Region r, Experiment e, Region rank_basis){

   CONDITION: summary(r,e).nr_remote_accesses>0;

   CONFIDENCE: 1;

   SEVERITY: remote_access_time(r,e) / duration(rank_basis,e);

}


Property remote_access_to_variable
   (Region r, Experiment e, String var, Region rank_basis)
{
LET
   VariableRemoteAccesses var_sum =
        UNIQUE({info IN summary(r,e).sums.accessed_variables
               WITH info.var_name==var});
```

```
IN
   CONDITION: var_sum.nr_remote_accesses > 0;

   CONFIDENCE: 1;

   SEVERITY:  var_sum.nr_remote_accesses * e.system.remote_access_time
              /duration(rank_basis,e);
}


Property multiple_transfer_of_same_data
        (Region r, Experiment e, String var, Region rank_basis)
{
LET
   VariableRemoteAccesses var_info =
        UNIQUE({info IN summary(r,e).sums.accessed_variables
               WITH info.var_name==var});
IN
   CONDITION: (Cond1)
                 (var_info.nr_remote_accesses * e.nr_processors > var_info.size
                 / e.system.cache_line_size*summary(r,e).nr_executions)
              OR
              (Cond2)
                 (var_info.nr_remote_accesses * e.nr_processors
                  > 0.5 * var_info.size / e.system.cache_line_size
                    * summary(r,e).nr_executions);

   CONFIDENCE: MAX( (Cond1)->1, (Cond2)->0.5 );

   SEVERITY:  var_info.nr_remote_accesses * e.system.remote_access_time
              /duration(rank_basis,e);
}


Property wrong_page_distribution_for_variable
   (Region r, Experiment e, String var, Region rank_basis)
{
LET
   VariableRemoteAccesses var_sum(String var,setof VariableRemoteAccesses info_set) =
        UNIQUE({info IN info_set WITH info.var_name==var});
IN
   CONDITION:
         EXISTS
            thr_sum   IN summary(r,e).thread_sums,
            thr_psum  IN var_sum(var,thr_sum.accessed_variables).page_sums,
            glo_psum  IN {s IN var_sum(var,summary(r,e).accessed_variables).page_sums
                          WITH s.page_no==thr_psum.page_no}
         SUCH THAT
               thr_psum.nr_remote_accesses!=0
            AND
               thr_psum.nr_remote_accesses==glo_psum.nr_remote_accesses;

   CONFIDENCE: 0.5;

   SEVERITY:  var_sum(var,summary(r,e).accesses_variables).nr_remote_accesses
              * e.system.remote_access_time/duration(rank_basis);
}
```

```
Property parallel_organization( Region r, Experiment e, Region rank_basis) {

   CONDITION:  typeof(summary(r,e).region)==ParallelRegion
                    AND summary(r, e).nr_executions > 1 ;

   CONFIDENCE: 1;

   SEVERITY:   summary(r,e).nr_executions *
               e.system.parallel_region_cost / duration(rank_basis,e);
}

END
```